Sebastian Grill, BSc

# Machine Learning Assisted Heat Detection in Dairy Cows

## Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Electrical Engineering and Audio Engineering

submitted to

**Graz University of Technology**

Supervisors

Univ.-Prof. Dipl-Ing. Dr. Franz Pernkopf[1]
Dipl-Ing. Dr. Christian Knoll[1]
Dipl-Ing. Dr. Tobias Rauter[2]

Institute for Signal Processing and Speech Communication
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Gernot Kubin

Graz, January 2021

[1]Institute for Signal Processing and Speech Communication, TU Graz
[2]smaXtec Animal Care GmbH

This document was compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

# Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

# Acknowledgments

I would like to thank both my supervisors at Signal Processing and Speech Communication Institute, Graz University of Technology, Franz Pernkopf and Christian Knoll for their mentoring during the entire course of this work. Throughout our many meetings they helped me gain a deeper understanding of neural networks and provided helpful input with architectural decisions, as well as guiding me through the organizational hurdles along the way. I am especially grateful that they took time out of their Christmas holidays to edit this thesis.

I would also like to thank my supervisor at *smaXtec*, Tobias Rauter, who was instrumental in making infrastructural decisions on the code surrounding the deep learning models, identifying many of my misconceptions early on and for his help in finding and fixing the occasional intricate bug or performance issue. His good natured sarcasm provided me with motivation when writing seemed arduous and I consider myself privileged to have been able to work with and learn from such a gifted engineer over the last couple of years.

Furthermore I would like to thank my classmate and friend Ludwig Mohr at the Institute of Computer Graphics and Vision, Graz University of Technology, for giving me a crash course in applied neural networks, introducing me to *pytorch* and our countless machine learning related virtual water cooler discussions that helped me understand many of the concepts involved and without a doubt greatly sped up my progress.

I would like to thank all people at *smaXtec* involved in making this thesis possible, especially Alexander Oberegger and Matthias Wutte. I would also like to thank all my colleagues at *smaXtec* for their outstanding camaraderie, professionalism, helpfulness and dedication to nurturing an amicable workplace atmosphere.

I would like to thank my parents and family, without whose unconditional support and encouragement I would never have endeavored to enroll at university.

Finally I would like to thank my partner Barbara, for her endless patience (which I have thoroughly tested), encouragement and support throughout my entire academic career but especially during the work on this thesis.

# Abstract

Agriculture is currently undergoing a rapid transformation, driven by digitization. One aspect is heat detection in dairy cows. Manual detection is prohibitively time consuming, especially in larger farms, but since the advent of the Internet of Things, it is has become possible to continuously collect time series of cow health parameters that enable automation.

Livestock farming features diverse processes and conventions all over the world, owed to different climate zones, farm sizes and local conditions, which in term is reflected in the cattle monitoring data. Conventional algorithms struggle to robustly detect heats from dairy biosignals out-of-the-box, instead requiring manual parameter adaptation on a case by case basis.

This thesis presents two machine learning models based on feed forward and recurrent neural networks respectively, in an attempt to improve generalization of detection. While ultimately their performance falls short of what is required by productive use, experiments performed revealed shortcomings in data labels, that, if addressed correctly, hold the potential for great improvements. Despite the lack of performance, the recurrent neural network model demonstrated that it was able to learn the underlying problem and could, with further improvements, achieve the desired outcome.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Smart Farming, the digitization of agricultural production, is an active research field. The aim of smart farming is to aid farmers and reduce their workload by providing data and insights.

In dairy farms, continuous monitoring of cattle can provide information about health and productivity of cows that otherwise would be prohibitively work intensive or even impossible to acquire. One topic of interest is the management of dairy cow reproduction, in particular, detection of heats. Cows only lactate for a limited amount of time after parturition, therefore dependable heat detection is necessary to ensure productivity and avoid prolonged periods in which the cow does not produce milk. Traditionally, farmers would spend time monitoring cows visually for increased activity. Globalization has led to increased competition, causing milk prices to decrease, making this practice economically inviable. Thus, there is an increased need for alternative ways of heat detection.

Data driven heat detection can offer a cost effective alternative to visual observation of cows. Continuous time series measurements of individual cow activity have long been postulated to be a feasible way of detecting heats [Kid77]. Additionally, there is evidence that suggests the same can be achieved with body temperature [Gas+15]. A robust and reliable data-driven system for heat detection must however consider the following aspects:

- Farming practices vary wildly depending on climate, vegetation, altitude, regional customs and local environment. Whereas on some farms, cows may be held indoors the entire year, on others animals will cover significant distances each day between a milking parlor and their pastures. This will result in a great variety of different activity patterns.

- The body temperature will be influenced by seasonality.

- Sicknesses can and and often will cause artifacts in temperature or activity or both. So can other environmental influences, like the presence of predatory animals or dogs, disturbances during nighttime due to an animal calving, failure of a water supply, etc.

- The data is heavily biased due to cows spending the majority of their time being not in heat. Typically a heat will last for 12-18 hours roughly every 21 days. In practice under typical conditions, for every time stamp labeled as heat, there are 230-250 timestamps labeled not as heat.

These issues mean that signal statistics will not only be non-stationary in time, but also vary greatly from farm to farm. Under these conditions, it is challenging to design traditional algorithms that will successfully cope with the full range of signal variability and still produce the desired performance without needing careful adaptation to each farm's individual properties. The advent of the Internet of Things (IOT) makes it possible to acquire an abundance of health related continuous time series data. In parallel to this development but also driven by the amount of available data and enormous advances in compute resources, machine learning methods have pushed the boundaries of classification performance in many fields during the last 10 years and often surpassed hand-made rule based systems by far. Given an ample amount of training data, machine learning models have demonstrated the capability to cope with adversities such as time variance and produce classification performance often surpassing that of humans.

The main objective of this thesis is therefore to develop a robust machine learning model prototype that is capable of reliably detecting heats in cows being held under diverse conditions. The raw data is provided by the company *smaXtec Animal Care GmbH* [Ani] and consists of individual cow's temperature and activity time series measurements as well as meta data.[1]

To be able to achieve robust heat detection, the following subtasks need to be accomplished:

[1] A detailed description of the dataset can be found in chapter 3.

- The data is persisted across several databases of different types. For preprocessing the data effectively, it needs to be acquired from these sources.

- The data needs to be cleaned. Since the data is acquired from an IOT sensor network, there is a real possibility of network outages or sensor failure causing data holes. Additionally, sensors have systematic measurement errors. These measurement errors can sometimes undergo temporal evolution as hardware ages. Data cleaning needs to account for these cases as much as possible and remove them from the dataset if specifications are exceeded.

- Before training a model, the data requires preprocessing. For example, in case of neural networks, this typically includes normalization of data and transforming categorical to numerical data via one-hot-encoding.

- We survey the state of the art in time series classification to identify promising techniques capable of achieving the main objective.

- A selection of promising techniques should be implemented in prototype form and assessed for their classification performance.

During the course of this thesis, the minor milestones were accomplished, culminating in the development of two different types of neural networks. Neither of the models achieved the primary goal of delivering robust classification performance, however closer inspection of the results revealed that the reason for the lack of performance lies in the dataset itself. Future work will focus on improving the dataset, which - after retraining - is expected to greatly improve both classifiers.

## Outline

Chapter 2 gives an overview of the current state-of-the-art in time series classification (TSC) and briefly discusses a few selected publications in the field that cover both traditional and machine learning based approaches. The second

part has a narrower focus on recent publications on the topic of automated heat detection. Chapter 3 contains information about the data used in this work. The sensor system used to measure and transmit the data is briefly explained, followed by a description of the available time series and meta data. The primary body of work is laid out in Chapter 4, describing how the data is gathered and how the pipeline developed for preprocessing works. It concludes with a description of both machine learning models. Chapter 5 presents the results of evaluations performed on the models described in chapter 4. A discussion of the results as well as problems identified with the data labels is contained in chapter 6. Finally, chapter 7 provides an outline of planned improvements as well as a general outlook.

# 2 Related Work

This chapter gives an overview of work related to this thesis by first looking at TSC both from the angle of traditional algorithms (section 2.1.1) as well as the field of deep learning (section 2.1.2). Section 2.2 then focuses on related work in the field of automated heat detection. Section 2.3 contains a summary of related works and briefly discusses the relevance for this thesis.

## 2.1 Time Series Classification

TSC is an active and broad research topic. As such, providing more than a mere superficial view on the field would be beyond the scope of this thesis. Nonetheless a few selected supervised approaches for TSC are discussed, first traditional[1] ones that are often used as a performance baseline in literature. Second, a few deep learning architectures are presented. These emerged mostly from the fields of computer vision or language processing and speech recognition, but have successfully been applied to TSC as well.

### 2.1.1 Traditional Approaches

#### Dynamic Time Warping & k-Nearest-Neighbors

k-nearest-neighbors (k-NN) clustering based on dynamic time warping (DTW) as distance measure is frequently used as a baseline to judge the performance of TSC algorithms [Bag+17; Ism+19].

[1] The distinction between deep learning and traditional algorithms is not very clear in literature. For the purpose of this thesis, when manual feature engineering is involved, an algorithm is said to be not of the deep learning type and vice versa.

The DTW algorithm works by establishing a distance matrix $\boldsymbol{M}$ between all points $a_i$ and $b_i$ of two time series $\boldsymbol{a}$ and $\boldsymbol{b}$. The distance measure can be any applicable distance, like Mahalanobis, etcetera. In this example we will use the squared euclidic distance.

The elements of the distance matrix are defined as

$$M_{i,j} = (a_i - b_j)^2 \text{ with } i \in \{0, I\},\ j \in \{0, J\}. \tag{2.1}$$

We now introduce a path as a set of points, i.e. $P = \{p_1, p_2, ..., p_k, ...p_K\}$, where each point consists of a tuple of coordinates $p_k = (i_k, j_k)$. A valid path must satisfy the conditions $\forall k \in [1, K-1], 0 \leq i_{k+1} - i_k \leq 1$ and $0 \leq j_{k+1} - j_k \leq 1$.

For each item in $\boldsymbol{M}$, the algorithm then calculates an item in the path distance matrix $\boldsymbol{D}$ where each item $D_{i,j}$ is the minimum possible accumulated sum of matrix items following a valid path from $M_{1,1}$ to item $M_{i,j}$. The optimum warping distances are then found by backtracking, starting at $D_{I,J}$ and following the smallest adjacent elements in $\boldsymbol{D}$ to get to $D_{1,1}$. This way, the warping distances are identified that effectively minimize the distance between $\boldsymbol{a}$ and $\boldsymbol{b}$.

This algorithm has complexity $O((I+1)(J+1))$, with the assumption $I + 1 \approx J + 1 \approx n$ this can be further simplified to $O(n^2)$. Due to this rather high complexity, normally restrictions are placed on the maximum amount of warping to reduce runtime. More recently however, optimizations were proposed to reduce the complexity of DTW [GS18; TD20].

Additionally, many publications put focus on further improving upon the robustness of DTW as a measure of time series similarity. A shortcoming of DTW is identified in [JJO11] as the lack of regard to phase difference between points. The authors propose what they term "weighted dynamic time warping", where differences with high phase difference are penalized through high weights. Another approach to improve robustness is proposed in [Mar09], in the form of time warp edit distance. The similarity is quantified by measuring how many "edit operations" need to be performed by the algorithm to transform one time series into another. A similar approach was taken in [SAD13], however the approach to transforming one time series into another was accomplished by a different set of operations ("move, split and merge").

Once a time-warp independent distance has been calculated, the k-NN algorithm [CH67] with $k = 1$ is used to classify time series. In general, k-NN works by assigning a sample the same class as the majority of its $k$ neighbors, where "neighbors" are the $k$ samples closest in the feature space. The literature surveyed does not qualify why $k = 1$ is widely used as the benchmark, but presumably DTW is a strong feature resulting in a trivial classification problem that does not make a majority vote (i.e. $k = 3$ or $k = 5$) with it's associated increased runtime complexity necessary.

### Collective of Transformation-based Ensembles and Hierarchical Vote Collective of Transformation-based Ensembles

In [Bag+15] a technique termed collective of transformation-based ensembles (COTE) is presented. As the name implies, it is an ensemble technique, i.e., 35 classifiers are trained on several transformed feature space representations of the input time series. The transformations used include the shapelet transform [YK09], the "periodogram transform" as obtained by performing the discrete Fourier transform (DFT), and an autocorrelation-based transform. These are then used as feature for classifiers in a heterogeneous ensemble, where classifiers are combined via weights obtained in cross-validation. The classifiers used are k-NN, naive Bayes, C4.5 decision tree, support vector machines with linear and quadratic basis function kernels, random forest, rotation forest and a bayesian network. This ensemble type is termed flat-COTE to distinguish it from other ensemble-based methods also studied in the publication. The authors propose a significant performance advantage over other approaches for flat-COTE tested on 72 data sets, most of which are part of the UCR time series database [Dau+18].

Hierarchical vote collective of transformation-based ensembles ( HIVE-COTE) [LTB16] is a further improvement of COTE that adds two additional transformation types, two additional classifiers and a hierarchical voting system. Several publications name HIVE-COTE as current state-of-the-art regarding TSC performance [Bag+17; Ism+19].

Classification performance aside, both flat-COTE and HIVE-COTE are immensely complex to train. The high number of hyperparameters and classifiers makes this approach impractical for many applications. Additionally, both

methods rely on the shapelet transform, which is stated to have runtime complexity $O(n^4m^2)$ where $m$ is the number of time series in the dataset and $n$ is their length [Bag+15], which is extraordinarily high.

Given these drawbacks, research continues to look for methods that are both easier to train and have a smaller computational resource footprint.

### 2.1.2 Deep Learning Approaches

#### Introduction

**Multilayer Perceptron** A Multilayer Perceptron (MLP), also referred to as feed-forward neural network or fully connected neural network is the most primal form of neural network and is rooted in research dating back as far as the middle of the 20th century [Ros60]. It consists of an input layer, an output layer and a variable number of hidden layers in between. Each layer can contain a variable number of neurons (figure 2.1). The layers in a MLP are typically fully connected layers, where each neuron of one layer is connected with all neurons of the subsequent layer.

Mathematically, a single layer network maps an input $\boldsymbol{x}$ onto an output $\boldsymbol{y}$ via the relation

$$\boldsymbol{y} = f(\boldsymbol{x}, \boldsymbol{\theta}). \tag{2.2}$$

$\boldsymbol{\theta}$ is the set of parameters that specify the model [GBC16]. Each neuron performs the operation

$$y = g(\boldsymbol{w}^T\boldsymbol{x} + c) \tag{2.3}$$

with $\boldsymbol{w}$ being the weight vector, $c$ being the bias and $g$ being a nonlinear activation function. Figure 2.2 illustrates this relation.

Historically, a number of functions have been used as activation functions like the logistic function, also known as sigmoid function or the tanh function.

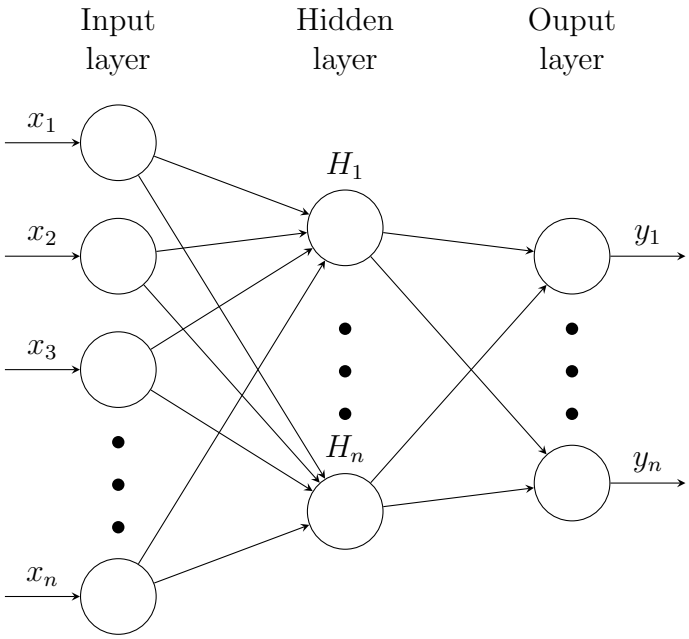


Figure 2.1: General layout of a Multilayer Perceptron.

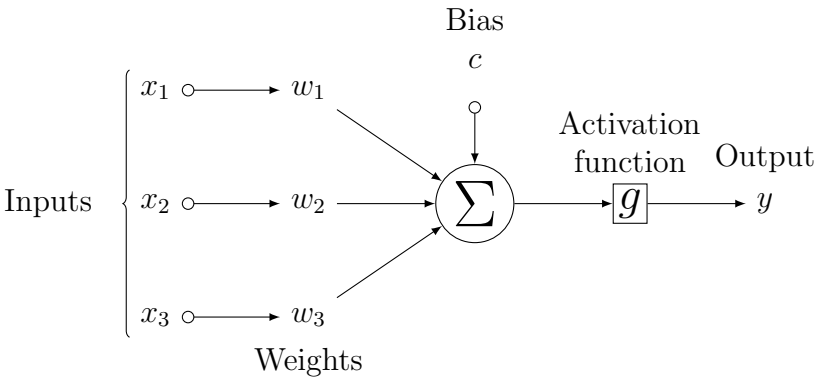


Figure 2.2: Schematic of a neuron in a fully connected neural net.

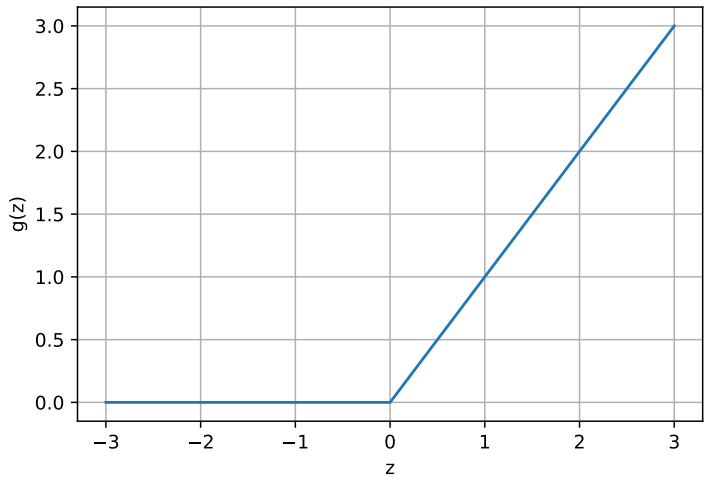


Figure 2.3: The rectified linear unit activation function.

More recently those have been superseded by the rectified linear unit (ReLU) function (equation (2.4), figure 2.3) [Jar+09; NH10; GBB10].

$$g(z) = \max\{0, z\} \tag{2.4}$$

Figure 2.1 illustrates a MLP with a single hidden layer. Using equations (2.3) and (2.4), when moving from a single neuron to a layer of neurons, introducing $\boldsymbol{W}$ as the weight matrix and $\boldsymbol{c}$ as the bias vector, the output $\boldsymbol{h}$ of the hidden layer becomes

$$\boldsymbol{h}(\boldsymbol{x}) = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c}). \tag{2.5}$$

The final output $\boldsymbol{y}$ becomes

$$\boldsymbol{y} = g_y(\boldsymbol{W}_y^T\boldsymbol{h}(\boldsymbol{x})). \tag{2.6}$$

$\boldsymbol{W}$ has a subscript here to distinguish it from the weight matrix of the hidden layer. Typically, each layer in a MLP has its own independent weight matrix.

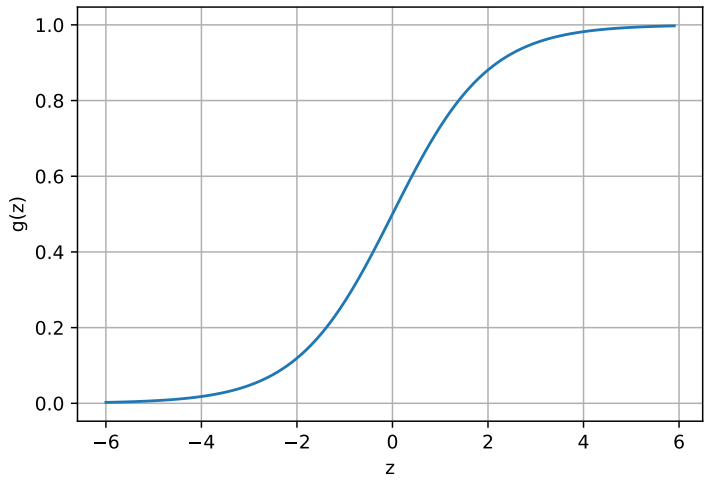


Figure 2.4: The logistic function.

Similarly, the subscript of the activation function $g_y$ emphasizes that the output layer typically uses different class of activation function than all of the hidden layers. In case of binary classification, a sigmoid function, also known as logistic function, is a common choice. The logistic function is depicted in figure 2.4 and is defined according to (2.7). In case of multi-class classification, the softmax activation function is preferred over the logistic function as activation function of the output layer.

$$g_y(z) = \frac{e^z}{e^z + 1} \tag{2.7}$$

The number of neurons in a neural net influences the complexity of the input-output relation that can be provided by the model. A larger number of neurons results in a larger set of functions that the model can provide, but since the number of parameters increases, it also makes it harder to train. The number of parameters can not only be increased by increasing the number of neurons in a single hidden layer, but also by increasing the number of hidden layers, which is often referred to as the *depth* of a network. In case of deeper networks, the

layers are chained together. Let equation (2.8) be the output of the $m$-th layer in a multi-layer network, given an input $\boldsymbol{z}$ (analogous to equation (2.5)).

$$\boldsymbol{h}_m(\boldsymbol{z}) = g(\boldsymbol{W}_m^T \boldsymbol{z} + \boldsymbol{c}_m) \tag{2.8}$$

The output $\boldsymbol{y}$ of the entire network with $M$ hidden layers for an input $\boldsymbol{x}$ is then given by equation (2.9).

$$\boldsymbol{y} = g_y(\boldsymbol{W}_y^T \boldsymbol{h}_M(\dots \ \boldsymbol{h}_2(\boldsymbol{h}_1(\boldsymbol{x})))) \tag{2.9}$$

A relatively recent optimization for the training process of deep learning models is batch normalization [IS15]. During training of a network, first the input is propagated through the model. Then the output is compared to the desired output to compute the error that was made. Afterwards, the error gradients are computed by backpropagation through the network. In this step, the weights are adjusted via the gradients under the assumption that only a single layer is adapted. However, during each adaptation, all layer weights are actually adjusted together, resulting in unexpected effects [GBC16]. To sidestep this problem and achieve faster learning, [IS15] propose normalizing the output of each layer according to

$$\boldsymbol{y} = \frac{\boldsymbol{x} - \mathrm{E}(\boldsymbol{x})}{\mathrm{Var}(\boldsymbol{x}) + \boldsymbol{\epsilon}} * \boldsymbol{\gamma} + \boldsymbol{\beta}. \tag{2.10}$$

The parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learned and enable the output to have independent $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$. This seems counterproductive after normalization, but results in an output that can represent the same family of functions as without normalization but with different, improved learning dynamics for the entire system [GBC16]. Learning of $\boldsymbol{\gamma}$, $\boldsymbol{\beta}$, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ happens via running statistics over all input values in the training phase.

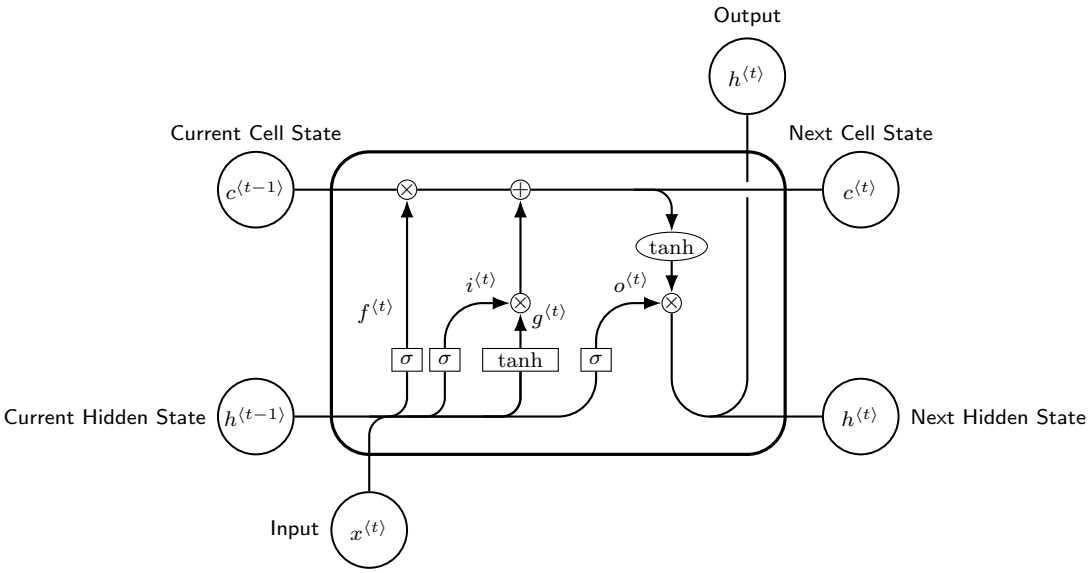


Figure 2.5: The internal structure of a LSTM cell, illustrating the position of the various gates.

**Long Short Term Memory** Long short term memory (LSTM) networks (figure 2.5) were first proposed in 1997 [HS97] but have received significant attention only much more recently. With advances in hardware, ultimately they have become extremely successful in fields like speech recognition and machine translation [GBC16].

LSTMs are a type of recurrent neural networks (RNN) and as such were specifically developed to process sequences of values where the order is important context [GBC16]. In contrast to purely feed forward networks (FFN) such as MLPs or feed forward convolutional neural networks (CNN) like fully convolutional networks (FCN) or residual networks (ResNet), RNNs are not memoryless systems. This means that not only the current input is considered in forming the output, but instead a hidden state is also considered that is formed by the network depending on previous inputs (figure 2.6). How the hidden state is formed and updated, depending on the input, is learned by the RNN. The graph of the RNN can be unrolled in the time domain (figure


Figure 2.6: The graph of a basic FFN compared to a basic RNN.

2.7). With this display, the dependency of the output at each time step on the hidden state of the previous time step becomes more explicit.

RNNs are trained via a process called back-propagation through time (BPTT). Given a significant length of the input sequence $\boldsymbol{x}^{\langle t \rangle}$, this process makes the RNN act like a very deep network. Similar to deep FFNs, earlier RNN structures suffered from what is called diminishing gradient. This means that during backpropagation, passing through many layers (or in the case of RNNs many recursions) the error gradient tends to vanish (or more rarely, explode). Without meaningful gradient information, the weights do not get adapted. This made previous generations of RNNs hard or even impossible to train properly.

The novelty of LSTM is the introduction of several gates, with the intent of giving the network the capability to learn to control when and how the internal state is updated and how much of the internal state versus the input is propagated into the output. As such, the cell itself dynamically controls its own memory, based on what it has learned during training and the input. The purpose of this mechanism is to enable the gradient to "flow" over longer periods of time without either vanishing or exploding [GBC16].

The internal structure of a cell with its four gates is illustrated in figure 2.5. Equations (2.11) through (2.16) describe the update mechanism during one time step, where $\boldsymbol{h}_t$ and $\boldsymbol{c}_t$ are the hidden state and cell state, $\boldsymbol{x}_t$ is the input, $\boldsymbol{i}_t$, $\boldsymbol{f}_t$, $\boldsymbol{g}_t$ and $\boldsymbol{o}_t$ are the input, forget, cell and output gates, all at time $t$. $\sigma$ is the sigmoid function and $\circ$ the Hadamard product (element-wise multiplication). $\boldsymbol{W}$ are weight matrices and $\boldsymbol{b}$ are bias vectors.

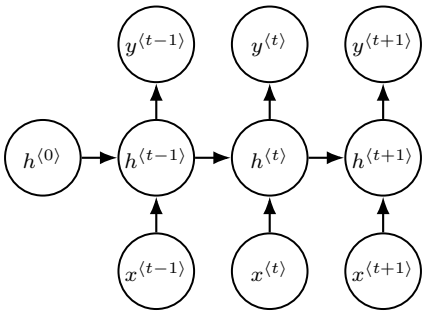


Figure 2.7: The same graph as in figure 2.6 but unrolled in the time domain over three time steps.

$$\boldsymbol{i}_t = \sigma\big(\boldsymbol{W}_{ii}\boldsymbol{x}_t + \boldsymbol{b}_{ii} + \boldsymbol{W}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{hi}\big) \tag{2.11}$$
$$\boldsymbol{f}_t = \sigma\big(\boldsymbol{W}_{if}\boldsymbol{x}_t + \boldsymbol{b}_{if} + \boldsymbol{W}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{hf}\big) \tag{2.12}$$
$$\boldsymbol{g}_t = \tanh\big(\boldsymbol{W}_{ig}\boldsymbol{x}_t + \boldsymbol{b}_{ig} + \boldsymbol{W}_{hg}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{hg}\big) \tag{2.13}$$
$$\boldsymbol{o}_t = \sigma\big(\boldsymbol{W}_{io}\boldsymbol{x}_t + \boldsymbol{b}_{io} + \boldsymbol{W}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{ho}\big) \tag{2.14}$$
$$\boldsymbol{c}_t = \boldsymbol{f}_t \circ \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \circ \boldsymbol{g}_t \tag{2.15}$$
$$\boldsymbol{h}_t = \boldsymbol{o}_t \circ \tanh(\boldsymbol{c}_t) \tag{2.16}$$

Figure 2.5 shows that the output of LSTM is its internal hidden state $\boldsymbol{h}^{\langle t \rangle}$, which is a vector with the same dimension as the width of the LSTM layers. Since the objective is both for LSTM to have a certain width as well as having the entire model return class probabilities for all classes in the label data, the hidden state has to be somehow mapped onto the class space. This can be achieved by applying a fully connected layer with sigmoid activation[2] onto the output of LSTM (equation (2.17), figure 2.8).

$$\boldsymbol{y}^{\langle t \rangle} = \sigma\Big(\boldsymbol{W}_y^T \boldsymbol{h}^{\langle t \rangle}\Big) \tag{2.17}$$

[2]If the problem is multi-class and not binary classification, a *softmax* activation would be preferable over the *sigmoid*.

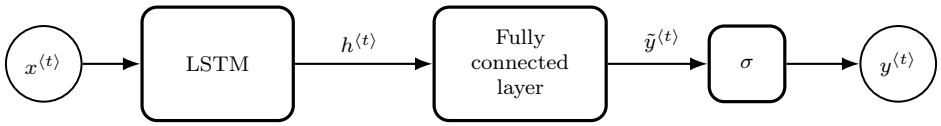


Figure 2.8: Blockchart of the entire LSTM system when used for classification.

**Convolutional Neural Networks** Much like LSTMs, CNNs are an old concept, dating back to works of Alexander Waibel and Yann LeCun in the late 1980ies. Similar to LSTMs, CNNs too received much more attention starting with the second decade of the 21st century, owing to progress in computer hardware.

As the name indicates, a CNN layer employs an operation termed *convolution*. The convolution of two signals $f$ and $g$, indicated by the operator $*$ is defined as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau. \tag{2.18}$$

Similarly, the discrete convolution of two discrete time signals $f$ and $g$ is defined as

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[n-m]g[m]. \tag{2.19}$$

In general, CNNs are often used in image processing, where using higher-dimensional convolution is desired. For applications with one-dimensional time series signals, the one-dimensional convolution (equation (2.19)) however is sufficient. A 1-d convolutional layer uses the convolution operation to *convolve* each input signal with one or more *kernels* $\boldsymbol{w}$ of weights, which are learned parameters. The length of the kernel $\boldsymbol{w}$ is a hyperparameter, analogous to the width of a layer in a MLP. The output of a convolutional layer need not be the same dimension as the input, it can also be a multiple of the input size, in which case multiple kernels are used with each input to produce multiple outputs. The output size of a layer too is a hyperparameter. Other hyperparameters include whether the kernel is *dilated*, i.e. there is spacing between kernel elements and

the *stride*, which indicates if the kernel slides continuously or "skips" a number of samples. The output of a convolutional layer is often referred to as the *feature map*.

CNN architectures often consist of convolutional layers combined with *pooling*, which is an operation that reduces the dimensionality through certain operations (e.g. taking the maximum of a certain window). Recent research in time series classification however has focused on architectures that are fully convolutional, i.e. they do not include pooling operations, instead stacking a varying number of convolutional layers on top of each other.

The motivation for CNNs with regard to TSC comes from two special properties [GBC16]: Parameter sharing and equivariant representations.

Parameter sharing means that a parameter is used more than once. Recall the introduction to MLPs, where the length of temporal "memory" of the model determined the size of the input layer. This means that for each input feature, the weight matrix of the first hidden layer holds a vector of separate weights that are only used once. In a convolutional layer, the length of a kernel $\boldsymbol{w}$ is independent of the length of the input. In a typical application, $\boldsymbol{w}$ is much shorter than the input signal, meaning the parameters are shared for all samples of the input signal. This lets a CNN be much more flexible with regard to the number of its parameters.

Equivariance means that the output is invariant to time shifts. Shifting an input sequence by a number of samples will result in the output sequence being equally shifted but otherwise similar.

These are both desirable properties of a model when performing TSC.

### Related Publications

**Fully Convolutional Network and Residual Network** In [WYO], the authors implemented three neural-network-based deep learning architectures for TSC, a MLP, a FCN [SLD17] and a ResNet [He+16] and compared their performance on a subset of the UCR dataset [Dau+18].

For the FCN, three convolutional layers were used, each coupled with batch normalization and followed by a ReLU activation function (figure 2.9). The

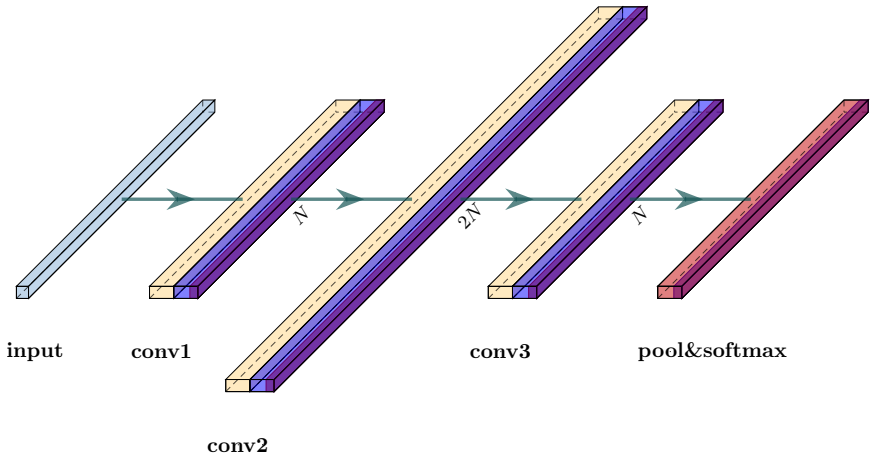


Figure 2.9: Fully Convolutional Network architecture as proposed in [WYO]. Yellow bars indicate convolutional layers, dark blue are batch normalization steps and violet indicate ReLU activation functions. The final two layers are global average pooling and a softmax activation function.

convolutional layers use [128, 256, 128] sized filters respectively, with convolution kernels sized [8, 5, 3]. The output of the final convolution layer is fed to a global average pooling layer and into a softmax layer.

In case of ResNet, the same convolution layers with batch normalization and ReLU activation are used as for the FCN. The main difference is that ResNet contains many more layers. ResNet consists of 3 blocks, which are each made up of three consecutive layers of convolution, normalization and activation (figure 2.10). The first block uses 64 filters, while the latter two blocks use 128. Same as in the FCN, convolution kernels are sized 8, 5 and 3, but for respective blocks instead of layers. Another major difference of ResNet compared to FCN is the usage of skip connections that bypass each convolution block. The motivation for this architecture is to reduce the impact of the vanishing gradient problem that arises in very deep architectures by allowing gradient to flow directly through the skip connections.

FCN and ResNet both deliver state of the art performance on the chosen subset of the UCR dataset. The literature does not agree on which method is superior,

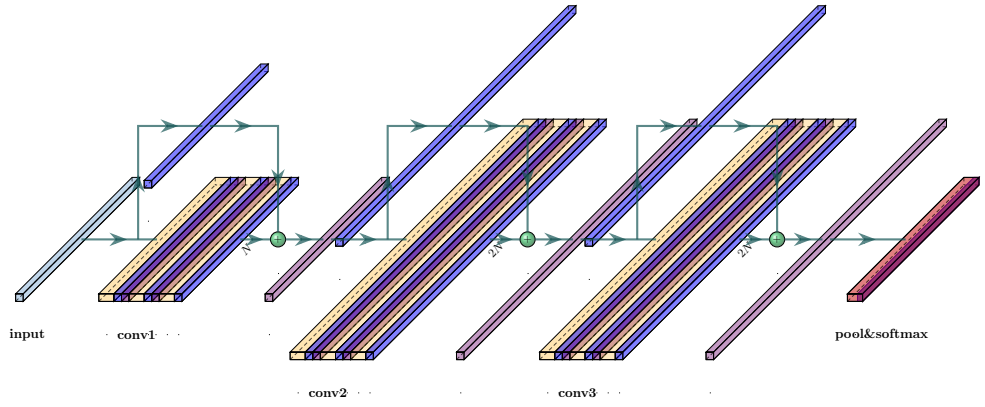


Figure 2.10: Residual Network architecture as proposed in [WYO]. Coloring scheme of units is similar to figure 2.9.

[WYO] found FCN to perform better while [Ism+19] suggest that ResNet offers better performance. The discrepancy is accounted by [Ism+19] to using a larger and more diverse subset of the UCR data.

**Long Short Term Memory** To improve the classification of the FCN proposed in [WYO], [Kar+17] used LSTM and FCN in parallel and concatenated their outputs before going through the final softmax layer. The LSTM uses the attention mechanism proposed in [Bah+16] to improve the learning of long term dependencies. With this model termed Attention Long Short Term Memory Fully Convolutional Network (ALSTM-FCN) the authors propose a significant improvement both over FCN and COTE.

An encoder-decoder based approach to classifying time series with LSTM was used in [Tan+16], also incorporating an attention mechanism. The authors compared their model to the results from [WO15], which they found their model outperforms.

Due to the structure of LSTMs, they have an advantage in processing time series data over memoryless architectures (MLPs, CNNs). No preprocessing is necessary to reshape input data to supply the network with information of past values, as the network efficiently saves this information in its hidden state. This

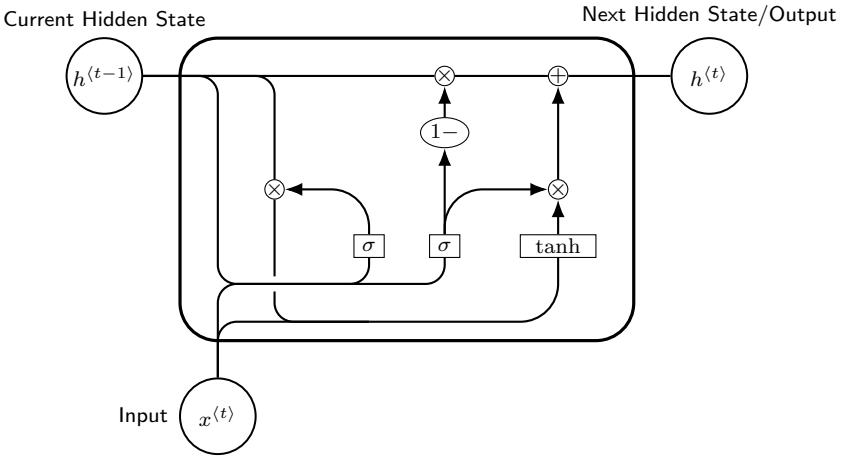


Figure 2.11: The GRU cell.

reduces the complexity of preprocessing when working on continuous data.[3] This advantage comes at a cost though, as training an RNN is a sequential process that can not be parallelized owing to its recursive nature. Despite its advantages for working with time series, LSTMs still struggle with long term dependencies [Tri+18].

**GRU** Gated recurrent units (GRU) are the second type of gated RNNs (together with LSTMs)(figure 2.11 shows a standard GRU cell). Its genesis lies in works such as [Cho+14; Chu+14], trying to simplify the complicated architecture of LSTMs by unifying the forget and the update gates [GBC16]. As such, the GRU exposes its state fully each time, as opposed to LSTM, where a gate controls the exposure [Chu+14].

The authors of [Che+18] used a slightly modified GRU in classifying multivariate times series with missing data from a medical database. The model is termed GRU-D where D stands for *decay*, which is a learnable parameter. Higher decay means that an input variable relies less on past values than lower decay. This model uses a mask internally to deal with missing data, additional inputs were time stamps of input values and time delta from the previous

[3]Section 4.1.3 explains the reqired reshaping process.

input. This modified version was compared to traditional GRUs and improved classification considerably. The authors attribute this to GRU-D learning to extract "informative missingness", that is correlations between missing data and certain classes.

With regard to performance of GRUs versus LSTMs, [Chu+14] did a comparison in the field of speech and music signal modeling. The conclusion was that no gated RNN held a definitive performance advantage over the other, instead performance was found to be highly application/data dependent.

**Transformer** The architecture termed transformer [Vas+17] is a very recent development, fueled by the intention to further improve upon the performance of LSTMs while doing away with its shortcomings, namely the need for strictly sequential computation, which does not allow for parallelization. The original work uses an encoder-decoder type structure for use in machine translation tasks. In this capacity, the task of the encoder is to transform the input sequence into a language-agnostic representation that is sent to the decoder, where the representation is transformed back into the target language.

When classifying time series, the objective is not to map the input series onto a different type of output time series but a class label instead. For this task, the encoder-decoder scheme is not needed and the output of the encoder can be used directly instead. The encoder itself is made up of an input embedding stage that converts input tokens[4] to vectors. Depending on the type of input data, this step is also not needed for TSC. This is followed by positional encoding, which injects information about the relative position of tokens in a sequence, a multi-headed attention unit and a fully connected feed forward type unit. It is proposed in [Vas+17] that their approach with a self-attention unit is able to learn long term dependencies more effectively than gated RNN type models at lower computational complexity.

This technique was used in [RK20] to classify crop types from multivariate optical satellite sensor data time series and found similar performance compared to an LSTM approach. Time series analysis on clinical data with a self-attention

[4] In case of machine translation, a token would be a word. In case of TSC, it would instead be a scalar or vector.

based model was performed in [Son+18]. The publication reports improved performance over a LSTM baseline.

### 2.1.3 Summary

In this section, several neural network architectures for TSC were introduced, along with a few conventional algorithms. Afterwards, a couple of publications were presented that compared these methods for their TSC performance. While the conventional algorithms are still widely regarded as benchmarks if not state-of-the-art, they have either too high computational requirements (DTW + k-NN), are too complex for productive use or both (COTE & HIVE-COTE).

The neural network architectures offer an attractive alternative. Research tends to favor feedfoward architectures (FCN, ResNet, transformer) over RNNs due to their lowered training complexity. Both FCN and ResNet have demonstrated good performance on subsets of the UCR time series database. Some research suggests that transformer could offer similar or better performance, but there are no direct comparisons yet to draw any definitive conclusions.

## 2.2 Estrus Detection and Machine Learning

Estrus detection is still a niche field where most innovation is currently driven by privately held companies that do not publish research save for performance evaluations that do not go into any sort of detail about the models being used in the field. Information about the true state of the art is therefore rather hard to come by. As in other medical research, field trials to acquire data from live animals are costly, which leads to sample sizes often being small, which in turn diminishes the significance of conclusions being drawn from the research. To the best of my knowledge, I am not aware of freely available dairy cattle datasets that could be used to objectively compare different methods. Nevertheless, in this section an attempt is made to provide an overview over publicly available research.

A team of researchers equipped 10 cows with acceleration sensor collars of their own design [YHL13]. The data was smoothed and 7 features (3-axis acceleration,

the derivative in all 3 axes as well as the mean acceleration) were used for further processing. Via k-means clustering a training set was created that splits the data into 3 classes (lying down, medium active and highly active).

This training set was used to train a Support Vector Machine (SVM) to classify the data. Finally an activity index is calculated from a comparison of current to historic activity together with the trend of classified activity levels. The activity index is then used via thresholding to discriminate heat events. With this method the authors managed to correctly classify 4 cows as having had a heat, while 6 did not. Given the small size of the surveyed population and there being no information about the time length of the surveyed data, the significance of these results is hard to place.

Some interesting work is presented in [Fau+19], where the authors performed a study comparing several machine learning approaches on a rather small dataset of 18,000 time samples. The used algorithms were k-NN, elastic net, SVM, random forest, gradient boosted trees, local cascade and a shallow MLP as well as their own approach termed *local cascade ensemble*. The local cascade ensemble offered best performance in this scenario, closely followed by gradient boosting and significantly improving over the performance of local cascade.

Activity data from pedometers as well as milk progesterone levels from 58 cows were collected in [OCo+11], totaling in about 14 accumulated years worth of time series data. Hidden Markov Models (HMM) were used to build both univariate as well as bivariate models, using either activity or progesterone or both together. Labels were created by looking at effective artificial inseminations while ineffective inseminations were excluded from the data. The univariate activity model is reported to have 70% sensitivity and 14% error rate while the bivariate model performs slightly better. The relatively low sensitivity is accounted to silent estruses being part of the test data.

## 2.3 Conclusion on Related Literature

The survey of literature regarding estrus detection with machine learning methods did not yield much useful information. What little publications are available either did not use state-of-the-art methods, worked with very small

datasets or both. Furthermore, due to the unique nature of how data is acquired for the dataset used in this thesis (see chapter 3), as well as the much bigger size, the applicability of published approaches remains debatable.

Looking at the field of TSC in general, current research suggests that CNNs like FCN or ResNet are most promising candidates for application in estrus detection. Despite this, a decision was made to implement a MLP and a LSTM-based model. The MLP was chosen as a baseline and testbed for ease of implementation while getting the code base set up. The decision to go with LSTM instead of one of the CNN models was forced by the complexity of the implementation required to achieve efficient training with minibatch processing (section 4.2) when working with input sequences of varying lengths. Based on the expected performance, FCN and ResNet remain the prime candidates and will be implemented during work following up to this thesis.

# 3 Dataset

Thanks to advances in wireless sensors and the IOT, it is nowadays possible to acquire interruption-free time series of sensor measurements. The applications are manifold, from airplanes and trains to factories where predictive maintenance is rapidly becoming standard to reduce or even negate downtime of costly machines, drivetrains or powerplants. This technology is also making its way into agriculture, where it can be used to monitor animals, aiding in improving health and productivity. Section 3.1 gives some background into how the data is measured and what is measured exactly, while section 3.2 goes into detail on how the data is structured.

## 3.1 Background on the Origin of Sensor Data

### 3.1.1 Sensor Placement

There are several established ways of acquiring bovine health monitoring data. Most of these include collars fixed to the neck or other extremities of animals. Conversely, the data used in this work was acquired using a sensor that rests in the gastrointestinal system. The sensor in the form of a small tube, as seen in figure 3.1, is called a "bolus".

The rumen of cows is subdivided into several parts, the frontal oriented part closest to the esophagus being the reticulum. If objects beyond a certain density are ingested by cows, they tend to drift to the bottom of the reticulum and remain there indefinitely for the remainder of the lifespan of the animal. The bolus dimensions and density are chosen to facilitate this mechanism, as such when the bolus is applied via the esophageal tube, the peristaltic movement

Figure 3.1: Sensor in bolus format

of the rumen causes the sensor to eventually drift past the feedstuffs to the bottom of the reticulum, where it will remain, as seen in figure 3.2.

### 3.1.2 Sensor Measurements

Each bolus is equipped with 2 types of sensors, a temperature sensor and a 3-axis accelerometer.[1] The sampling period for the eventual time series is 10 minutes.

In the case of temperature measurement, this is ample temporal resolution, given that the temperature signal does not undergo rapid changes. In case of the acceleration sensor, this sampling rate would constitute severe under-sampling. The sensor measures with 100 Hz and aggregates all 3 axes over 10 minutes, with several steps of preprocessing to filter out earth's gravitation and enhance the signal-to-noise ratio (SNR) resulting in a unitless acceleration index between 0 and 100.

[1] The manufacturer offers another type of bolus that also measures pH value, but this data was not used in this work.

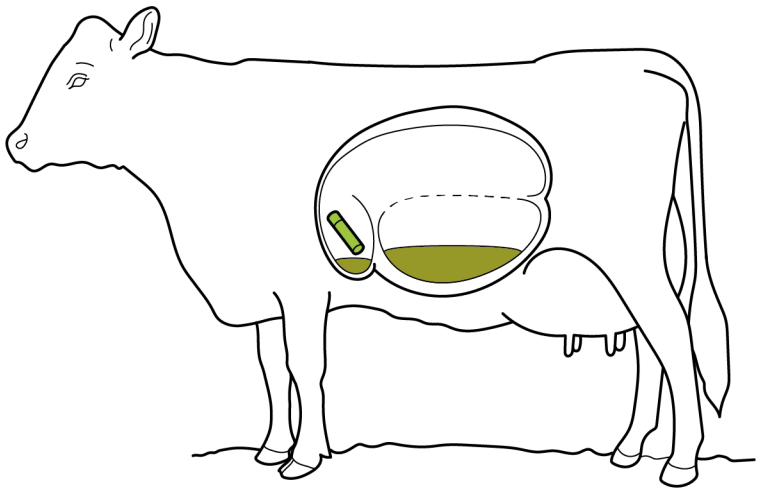

Figure 3.2: Schematic depiction of sensor placement

The preprocessing chain is proprietary and therefore can not be laid out exactly in this work. The low overall sampling rate is dictated by the limited power budget of the sensor and its internal battery. After preparation, the sensor periodically transmits its data via a gateway to a cloud server for storage and further processing.

## 3.2 Structure of Raw Data

The data can be divided into two subtypes, time series data and meta data. At the time of writing, the dataset contains 76852 animals with meta data, of which roughly 35000 have time series data available inside the period of consideration, starting with April 1st 2018 and ending with April 1st 2020.

### 3.2.1 Meta Data

The meta data holds all information pertaining to an animal that is not contained in the time series data in the form of a java script object notation

| datetime | activity | temperature |
|---|---|---|
| 2018-12-28 13:51:00 | 0.00 | 21.610001 |
| 2018-12-28 14:01:00 | 9.14 | 37.610001 |
| 2018-12-28 14:11:00 | 5.14 | 39.980000 |
| 2018-12-28 14:21:00 | 4.88 | 40.269001 |
| 2018-12-28 14:31:00 | 13.61 | 40.269001 |
| ... | ... | ... |

Table 3.1: Exemplary time series data

(JSON) file. Among a unique Identifier (ID), these are informations about the lactation state of the animal,[2] inseminations and pregnancy analyses performed by the farmer. Furthermore cows are typically held in groups. This affiliation is represented in the meta data through a group ID. Individual cow meta data is the primary source from which labels for the time series data are constructed.[3]

There is also a second level of meta data that holds information on an organization level. In general, an organization will represent a farm. Organization data is mostly relevant to the preprocessing stage, for example to localize timestamps from Universal Coordinated Time (UTC) to the respective timezone.

Sensitive data like country or actual farm affiliation were purged from the JSON documents for the purpose of information security.

### 3.2.2 Time Series Data

Time series data can be seen as a table of two columns holding 64 bit floating point numbers respectively, with the index in the time dimension being a string of format *YYYY-MM-DD HH:mm:ss* as outlined in table 3.1. For an outline of how the values are measured, see sections 3.1.1 and 3.1.2.

[2]The lactation status of an animal states which state of the lactation cycle the cow is in at one point. Possible statuses are pregnant, close to calving, lactating, dry, etcetera.

[3]The specifics of this process are outlined in section 4.1.3.

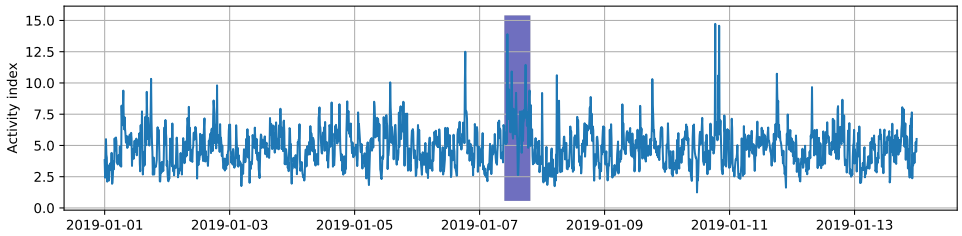


Figure 3.3: Example of an activity signal with heat annotation.

### Activity Data

Activity is a strong source of information for heat detection [Kid77; VV96; Fir+02]. Around the time of ovulation, activity shows a prolonged increase (see figure 3.3), the magnitude of which varies depending on outside influences, i.e. whether cows are held on pasture or indoors, material and surface texture of barn floors, etcetera. Depending on the circumstances, estrus related activity increases may be embedded in a considerable amount of noise. Conditions in New Zealand for example are typically such that animals are held on pasture and are driven to a barn twice a day, over a distance of several kilometers. These phases of increased activity can mask an estrus event quite effectively, making it necessary to apply measures such as incorporating the activity of the herd or daily activity patterns into processing to enhance the SNR. Figure 3.3 contains two examples of significant activity spikes outside of the range labeled as heat.

### Temperature Data

Similar to the activity, estrus also influences the body temperature of cows [WBS58; Hig+19; Wan+20]. Trials at *smaXtec* to create a purely temperature based estrus detection system faced issues in practice. While there is definitely correlation with estrus events, temperature alone has low specificity due to some sicknesses creating similar temperature patterns. Figure 3.4 shows part of an exemplary temperature time series. Noteworthy are the downward spikes, which are caused by water intake of the animal, as well as the temperature

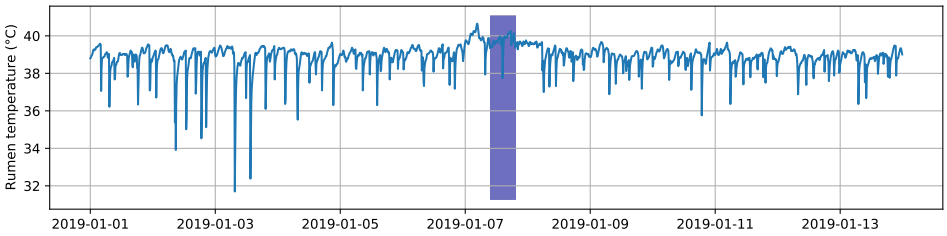


Figure 3.4: Example of a temperature signal with heat annotation from the same animal and time frame as the activity data in figure 3.3. The downward spikes in the signal are caused by the intake of cold water by the cow.

increase shortly prior to the heat annotation as well as the reduced water intake around heat. The water intake spikes are removed from the signal during preprocessing to increase correlation with the actual body temperature.

# 4 Methodology

This chapter documents the practical work undertaken for this thesis, starting with gathering of the required data from varying sources in section 4.1.1, the storage solutions chosen for various steps of the preprocessing in section 4.1.2 as well as the preprocessing itself in section 4.1.3, along with the technologies used in the process and closes with the machine learning approaches chosen and implemented in section 4.2.

## 4.1 Preprocessing and Data Representation

To enable usage of the data for training machine learning models, the data needs to be converted from its original raw form into something a machine learning model can "understand". In case of neural networks, this constitutes the form of strictly numerical vectors or matrices. Strings or categorical data must be converted to other representations. Normalization or standardization of inputs is often required, as is cleaning of the data. Figure 4.1 illustrates an initial concept of the approach envisioned for the preprocessing data flow and storage. Minor adaptations had to be made for practical reasons which are described in the following sections.

### 4.1.1 Data Gathering

The data in its raw form is distributed over several databases (see Figures 4.1 and 4.2). To speed up any further processing steps and limit the traffic load that acquiring large amounts of data puts on the databases and application programming interface (API) services, an intermediate raw database is created, that serves as the basis for all further processing.

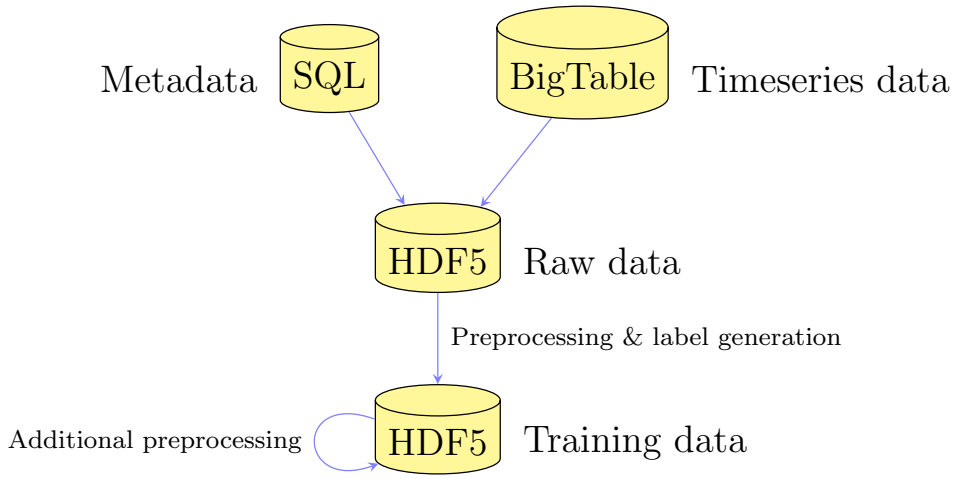


Figure 4.1: Initial concept for data gathering and preprocessing.

Meta data is acquired from postreSQL databases via a public representational state transfer (REST) API with an open source API client [sma]. The meta data is downloaded first because it is required to build the tree structure of the raw database, which is stored in HDF5. Additionally, the unique identification keys of animals are required for calls to the time series database. To be able to store the JSON files in HDF5, they need to be serialized first. For this the *json* component from the python standard library was used together with *h5py* [h5p] for writing to and reading from HDF5. Before the JSON documents are serialized, any fields that contain personalized information that could be used to identify a farm are deleted. The link to physical persons or farms can still be reconstructed via each organization's or animal's unique ID, but not without the information mapping IDs to their physical counterparts.

Acquisition of time series data from its *BigTable* database is accomplished using a proprietary database client provided by *smaXtec*. The client is based on an open source project implementing a time series store for *BigTable* [Wut]. Since the database client supports at most downloading 400 days of data at once, downloading the entire 2 years of data requires splitting the downloading of each animal time series into two separate calls. The client intermediately stores the downloaded data into comma separated values (CSV) formated text files,

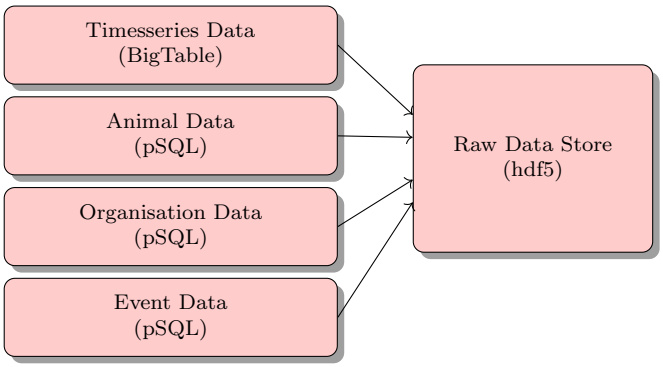


Figure 4.2: Schematic of the data acquisition step.

which - in a second step - are then parsed using *pandas* [pan]. Split parts are concatenated, the UNIX timestamp index is converted to a human readable ISO 8601 compliant datetime format (see section 3.2.2) and then serially written to HDF5[1] via *pandas'* API for *pytables* [pyt].

### 4.1.2 Data Storage

#### Raw Data Storage

The raw data storage uses HDF5. It is structured hierarchically, the top level is split into organizations, which are further split into individual animals (figure 4.3). Additionally, the database contains a mapping between animals and organizations to make lookup operations easier.

#### Temporary Intermediate Storage

Due to constraints imposed on the processing by available random access memory (RAM), along several points in the data preprocessing pipeline, data needs to be persisted into non volatile storage.[2] This was accomplished using the

[1] While HDF5 is a well established format, owing to its age it does not incorporate strategies for consecutively writing to a database, hampering write performance.

[2] The process is outlined in detail in section 4.1.3.

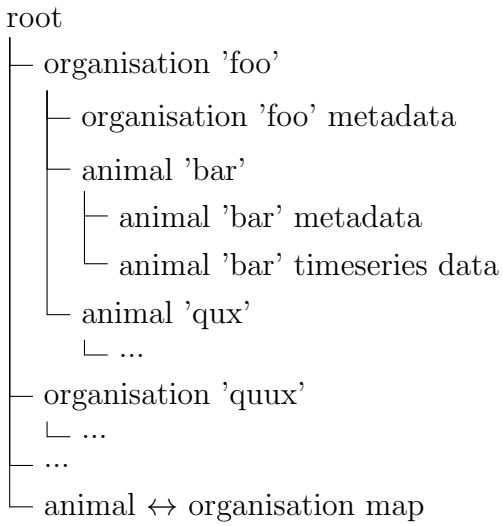


Figure 4.3: Hierarchy of data inside the HDF5 raw database.

*feather* format as offered by *PyArrow* [pya] for serialization and deserialization performance as well as good integration with *pandas*.

### Training Data Storage

Contrary to the initial concept in figure 4.1, HDF5 is not used as storage for the eventual training data. At the final stage, training data does no longer contain categoricals or any other non numeric types of data, which enables using database types optimized strictly for this purpose. *zarr* [zar] was chosen because it offers good read and write performance, the ability to read and write concurrently and compression of numerical higher dimensional array data as well as good integration with the parallel computing framework *dask* [das], which was used to speed up some of the processing steps needed for batch composition (section 4.1.3).

The eventual representations in *zarr* are sketched out in figures 4.4 and 4.5.

For the first representation, each row represents one sample, corresponding with one point in time that needs to be classified. The columns represent different

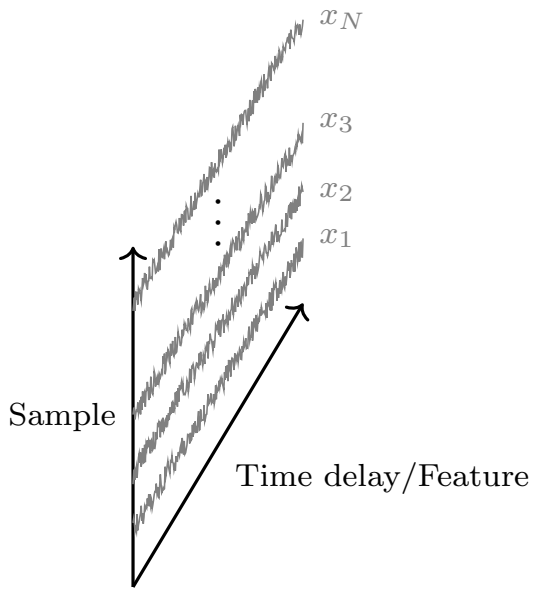


Figure 4.4: Arrangement of time series features in *zarr* for use with feed forward networks.

features, where the first column is reserved for label data. The second column contains the days that have passed since the last parturition of the animal. The actual time series data itself consists of 288 features, where the first feature represents the timestamp that should be classified and the remaining features are time delayed samples from the same time series, giving the model two days of "memory" of the time series.

The second representation is geared towards use with networks that do not need temporal context in the input, because it is provided by the internal state. This makes the time-to-space transformation unnecessary and saves a lot of redundancy in the data, freeing up resources that can be used to store additional features.

The main difference is that the data is not stored as one continuous tensor. Different amounts of time series data are available for different animals, which makes it impossible to store the data in a continuous array without padding

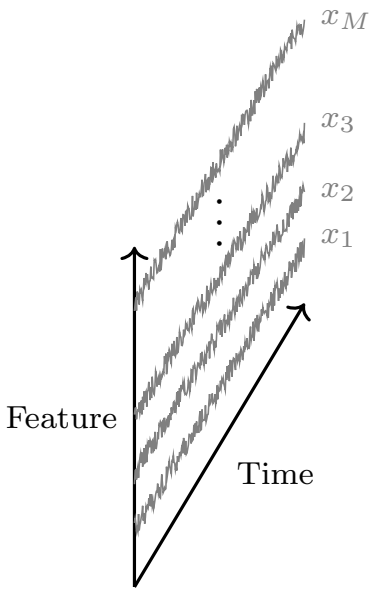


Figure 4.5: Arrangement of a single sample for use with a recurrent type of network.

all time-series to similar length.[3] Instead, the data is sharded along the animal key. This can be realized using *zarr*s tree-like data organization. The entire dataset is combined into a group that can be accessed like a dictionary via the animal ID as key. Each value is then a sample, consisting of the time-series of all available features and the annotation (figure 4.5).

### 4.1.3 Preprocessing

The task of preprocessing can roughly be divided into cleaning of the data, feature and label generation as well as standardization/normalization and batch preparation. Figure 4.6 contains an overview over the entire process.

[3]This would be possible, however simply training a network on zero-padded data would introduce bias and therefore worsen the eventual classification performance. Section 4.2.3 contains a detailed description of how this was achieved to enable batch processing.

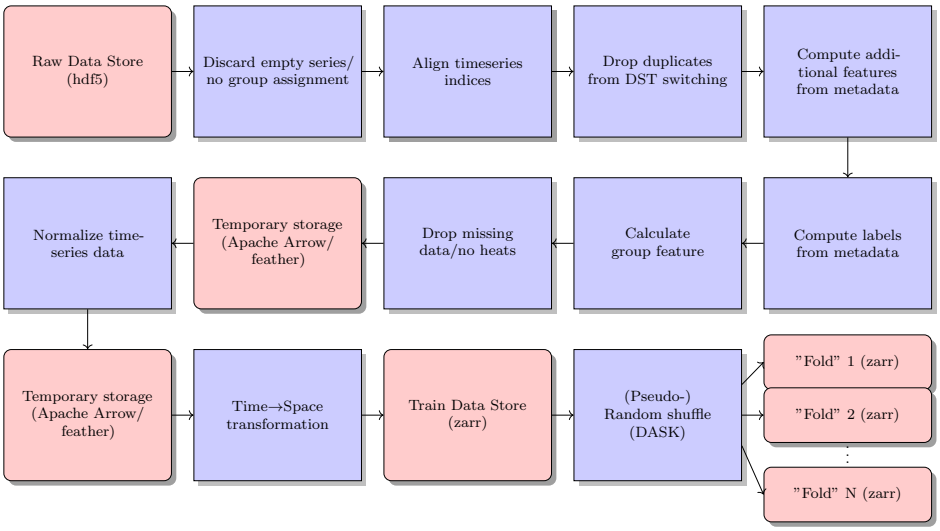


Figure 4.6: Overview of the preprocessing stage. Pink blocks represent (temporary) storage of data while violet blocks represent processing/transformation steps.

### Pipeline Architecture

The pipeline is arranged into four steps that are divided by (temporary) storage. The first step encompasses cleaning, label and feature generation, in the second step, time series are standardized, the third step rearranges the data to provide the machine learning model with a time window of data instead of a single sample and the fourth and final step splits the data into a test and a training set and shuffles samples so that when the machine learning framework loads batches of data to train the model, those batches are not biased by, for example, containing data from just a single animal.

Where possible, the pipeline makes use of parallel processing to utilize all resources provided to it by the host it is executed on. The first step uses python's *concurrent.futures ProcessPoolExecutor* class to fork as many workers as central processing unit (CPU) cores are available. Parallelization is achieved by processing organizations in parallel, with all animals in one organization processed in series.

The second step uses the *StandardScaler* from the *scikit-learn* library [skl], which performs two passes over the data. The first pass happens in a serial manner to fit the scaler parameters to each feature, then in the second step animals are scaled in parallel, once again using *ProcessPoolExecutor*. Writing to the first *zarr* database is done serially, as designing a parallel scheme was deemed out of scope for the purpose of this thesis. The last step of splitting and shuffling the data is once again performed in parallel using *dask* and *daskml*.

### Data Cleaning

Data cleaning does not happen in a single step, rather it is accomplished via two checks during the course of preprocessing. At the time of writing, cleaning is mostly focused on removing parts of data that are either too short[4] to be considered or lack some other critical information (e.g. no heats in the surveyed time span, no group information, etc.).

The first filter is situated immediately after data is loaded from the raw data store (see figure 4.1.3). If there is no time series data present for the observation period, the respective animal is immediately discarded and the pipeline continues with the next animal. The same is true for animals that are not assigned to a group.[5]

The second filter sits at the end of the first stage of the pipeline, after labels have been generated. For each animal we assess whether there are at least 30 days of recording and at least one heat annotation present in the data. If not, the animal is discarded. This step is necessary as group feature calculation may reduce the amount of data available for each animal, so that these criteria may be missed in the first step.

### Feature Generation

The pipeline generates the following four features:

[4] Animals with less than a month of time series data were not considered. The motivation is to exclude the possibility introducing bias because of incomplete temporal context. In the majority of cases in production, ample temporal context is available.

[5] Designing a model that can cope with missing group information is in principle desirable, but out of scope for this thesis.

| Index | | | | Features | Labels |
|---|---|---|---|---|---|
| Organisation ID | Group ID | Animal ID | Datetime | Feature A | Label A |
| abcd | defg | ijkl | 2018-1-1T00:00:00 | 42 | True |
| | | | 2018-1-1T00:10:00 | 42 | False |
| | mnop | qrls | 2018-1-1T00:00:00 | 15 | False |
| . | . | . | . | . | . |
| . | . | . | . | . | . |
| . | . | . | . | . | . |

Table 4.1: Schematic of the structure of the data and its index inside the *pandas DataFrame* in the preprocessing pipeline.

- The Days in Milk (DIM),
- the "group feature",
- the "heat feature" and
- the temperature with spikes from water intake removed.

The DIM is the number of calendar days that have passed since the last parturition. The reason adding this as a feature is that the model could learn to distinguish DIM with higher probability of heat from ones with lower probability, such as directly after parturition. Calculation of DIM was achieved using code provided by *smaXtec* that calculates this out of meta data[6] once for each calendar day, then using a frontfill strategy to fill in the timestamps in between.

The "group feature" is the mean of the activity index for each timestamp of each animal in a respective group. If the group is of sufficient size, taking the mean of the activity should suppress fluctuations of individual cows as the mean converges against the expected value of the activity, while containing environmental influences that affect all animals.[7] Computing the mean is straight forward using the *pandas DataFrame* format. Transforming the *DataFrame* to enable this calculation however is a bit more involved. Table 4.1 illustrates the format of the data before this calculation. It should be noted that at the beginning of the pipeline the datetime index is aligned to calendar 10 Minute intervals. This reduces the complexity and RAM footprint of several

[6] This calculation is in principle trivial, counting only days since the most recent previous parturition, however certain edge cases such as aborts have to be considered.

[7] An example for such an influence would be the group being driven to and from the barn for milking.

| | | | **Animals** | | | |
|---|---|---|---|---|---|---|
| | | | ijkl | | qrls | |
| **Index** | | | **Features** | **Labels** | **Features** | **Labels** |
| Organisation ID | Group ID | Datetime | Feature A | Label A | Feature A | Label A |
| abcd | defg | 2018-1-1T00:00:00 | 42 | True | nan | nan |
| | | 2018-1-1T00:10:00 | 42 | False | nan | nan |
| | mnop | 2018-1-1T00:00:00 | nan | nan | 15 | False |
| . | . | . | . | . | . | . |
| . | . | . | . | . | . | . |
| . | . | . | . | . | . | . |

Table 4.2: Schematic of the structure of the data after unstacking of the animal ID index.

calculations in the pipeline while introducing only minor sampling errors. To take the mean of each animal for each group, the table has to be transformed by unstacking the animal ID from the row index and making it a column index (table 4.2). The mean is then calculated considering only groups with more than 5 cows, inserted as an additional column to each animal, then the animal ID is once again stacked to the row index and the rows containing "Not a Number" (NaN) are dropped.

With the "heat feature", an attempt was made to compute a simple distance based metric between the single cow activity and that of its group. This should indicate whenever the behavior of one specific cow deviates from the behavior of the group. It is calculated by taking the difference between the cow and group activities, then applying a sliding window sum operation with a window length of 12 samples.

Temperature inside the rumen is influenced by several factors, such as actual body temperature, fermentation heat and to a large degree frequency and volume of water ingestion together with the temperature of the ingested water itself (see also figures 3.4 and 4.8 respectively). Since body temperature is the desired information, the spikes caused by water intake should be removed to make inference easier for the eventual machine learning model. The algorithm implemented in this work performs two passes over the data. Each time, a rolling statistic (median and $\sigma$) is calculated, from which an adaptive threshold is computed. Any value falling below the adaptive threshold is replaced by a long-term sliding mean (figure 4.7). The result of this filtering process can be seen in figure 4.8.

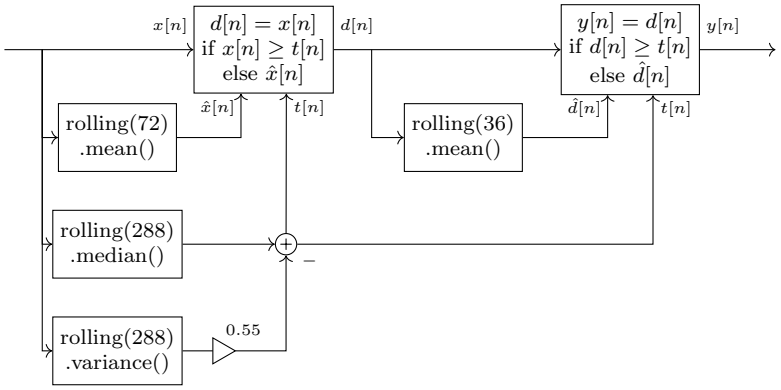


Figure 4.7: Block diagram of drink spike removal. Values in parenthesis state the length of the rolling window used in the respective computation. Delays required for causality have been omitted for simplicity. Window lenghts were determined empirically.

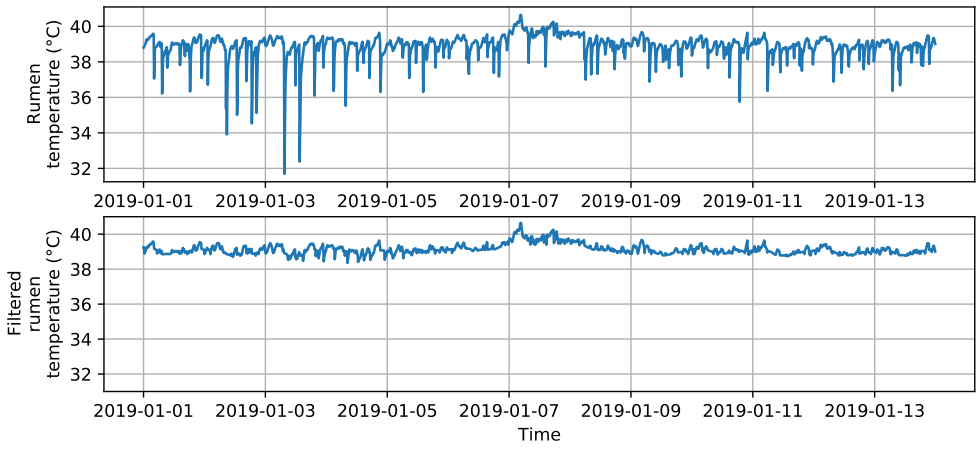


Figure 4.8: Rumen temperature signal before and after filtering with the algorithm depicted in figure 4.7.

### Label Generation

Label generation uses two types of data source, user input and labels provided by the preexisting classification system. In case of user input, insemination dates and manually performed pregnancy checks as well as calvings 270-290 days after a potential heat were regarded as gold standard. The autonomously generated labels by the classification system are first checked for periodicity. A single positive classification has a significant chance of being a false positive. In case there are two classifications that line up with the estrus cycle length within a tolerance of $\pm 3$ days, chances are good that both classifications are true positives.

For each class of labels, the pipeline gathers the relevant timestamps from meta data and applies the respective checks (for example, whether there are other heats before or after or whether there are calvings or pregnancy checks that confirm it as a heat). After this step, the local activity maximum is sought in a timespan of $\pm 12$ hours. The label is then shifted to this local activity maximum. The reasoning for this process is that users can not be expected to be that precise in their input of labels. To provide exact insemination information, the peak activity is an important reference point, from which the time of ovulation and optimum insemination time can be estimated. The timespan starting two hours prior to maximum and ending eight hours post maximum is then labeled as heat. Estrus itself is not a singular moment but lasts hours. Therefore with the labels an attempt is made to train the model to classify the time range relevant to farmers and not detect a single irregularity. This also reduces the class imbalance to some degree, although it remains quite significant.

After computation of all label classes, the label vectors are combined by applying an element-wise OR operation over all vectors, that is, the final label of a sample is set to one if at least one of the label vectors is one. Otherwise if all are zero, the final label is zero as well.

### Dataset Composition

Feed-forward neural networks such as the MLP used for this work do not possess any memory. As a consequence, any information needed by the network to perform its classification task needs to come from the input. In case of

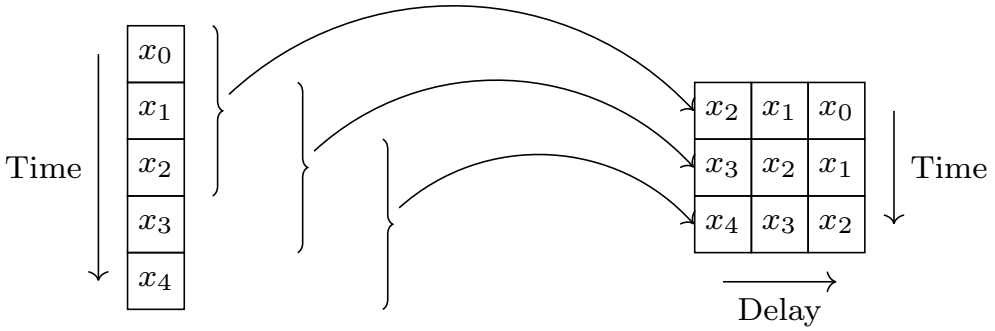


Figure 4.9: "Time-to-space" transformation.

time series classification, a significant portion of information is provided by previous values of the time series. The step in figure 4.6 titled "Time-to-Space transformation" deals with this requirement by transforming the time series in such a way that each sample is not a single value but instead a time slice (figure 4.9) of the value to be classified and its immediate past. This can be accomplished by duplicating the original time series $N$ times, where $N$ is the number of samples of past information that are to be passed to the model. After duplication, the $n$-th column is then time-shifted by $n$.

The performance of a machine learning model or any statistical model for that matter can not be evaluated with data that was used in training, because in such a case, the model could have "learned the data by heart" instead of learning an underlying pattern or ground truth. This would result in deceptively good performance during evaluation, while the model would then perform very badly on data it has not observed previously. This behavior is termed overfitting of training data, while a model that performs well on unseen data is said to generalize.

To forgo this problem, data is commonly split into parts of unequal size, one for training and another for evaluation. If not stated differently, experiments were performed with a split of $\frac{1}{3}$ of the data falling into the evaluation set and $\frac{2}{3}$ into the training set. Implementation-wise, the *train_test_split* method of the *dask.ml* library was used. This method also shuffles rows before performing the

split, so that any bias in the data that is a result of how it is ordered does not propagate into the test and training sets.

An optional final step for the preprocessing pipeline is the creation of multiple instances of the same training set by shuffling the data. Often when training a neural network, several passes are performed over the training data termed epochs. It may be beneficial to shuffle the training set between epochs. This option is available via a random permutation implemented by *dask*.

## 4.2 Machine Learning Models

This section documents the work undertaken to train and evaluate the machine learning models as well as the work to get the data loaded into *pytorch*.

### 4.2.1 Data Loader

There is a custom system for feeding data into a model provided in *pytorch*. This method is based on a class representing the data and another one that is called the data loader that samples from the data and passes it to the model, incorporating multi-process parallelism. This system offers a lot of flexibility because the user can write own implementations tailored exactly to the use case and the properties of the data. There is a strong interface that - if adhered to - ensures compatibility with *pytorch*.

With *pytorch* come two built-in ways of representing the data: The *map-style* dataset and the *iterable-style* dataset. In case of the map-style dataset, the data class provides a map from indices or keys to data samples. The data loader then performs (pseudo-)random access via these keys to build batches that are loaded into the model. This is a viable way of working with data where random access is performant.

In case of the training data format chosen for part of this work, random access on single non-adjacent samples becomes very slow because *zarr* splits the array into blocks that are individually compressed and stored. If random samples are accessed, all blocks that contain them have to be decompressed. To ensure optimal performance, it is desirable that the lowest possible number of blocks

has to be decompressed for any batch. In case of random access without using a cache, the number of blocks being decompressed would be impractically high,[8] resulting in bad performance.

Block size is configurable, but there is a tradeoff between block size and compression rate. Bigger blocks allow a more efficient compression at the price of taking longer to decompress and vice versa. Also each block is stored as a separate file. Making blocks too small can thus lead to issues with the number of files in a folder or partition (depending on the file system that is used). Due to these issues, it is impractical to use the map-style dataset together with a *zarr* database that combines the entire data into a single array. If, however, data can be with each animal having a separate array, this remains a viable solution.

The iterable-style dataset offers a second way of loading data. Instead of returning data for an index, an iterator over the data is returned. The data loader then repeatedly calls the iterators *next* method until it has enough samples for a batch. The difference at the database level is that samples are no longer chosen randomly, instead adjacent samples are loaded. This minimizes the number of blocks having to be decompressed because *zarr* implements an iterator with a built in block cache. Thus the current block is only decompressed once. Parallelism is achieved by having multiple iterators work on different slices of data.[9]

This approach has one major downside. That is, random sampling from the data is no longer achieved at the model training stage and one has to enforce the randomness when creating the dataset. This comes at comparatively little additional cost though, since data has to be shuffled prior to splitting between test and training dataset anyway (see section 4.1.3).

[8] The exact ratio of blocks and samples depends on blocksize, batch size and the size of the entire dataset. If the dataset is large compared to block and batch size, each sample would likely come from a separate block.

[9] *zarr*s previously existing iterator had no concept of iterating only over parts of the array. Since this would have hampered performance at startup of each data loader worker (the worker having to run the iterator to its starting point before being able to provide the requested data), a pull request was contributed to *zarr* that expands the iterator with this functionality.

There are several approaches in machine learning as to how a dataset can actually be consumed:

- Batch processing
- Sample by sample processing
- Minibatch processing

In batch processing, the entire dataset is consumed at once. This is the optimum with regard to stability of many optimization algorithms, however a prerequisite is that the entire dataset can fit into RAM or Video Random Access Memory (vRAM) in case a Graphic Processing Unit (GPU) is used instead of a CPU to speed up computation. Given the size of typical datasets required to train learning models such as LSTM of even modest dimensions, this is not an option.

On the other extreme of the spectrum there is sample-by-sample processing, where each input sample is loaded separately and the model is updated based on the error gradient computed on just one sample. While the memory footprint of this approach is much more manageable than batch processing, all numerical convergence advantages of batch processing are lost and as such, convergence may be much slower and the model may not converge against a solution of the same classification performance. Additionally, with this type of processing, a lot of parallelization potential is foregone, especially in GPU processing where a lot of pipelines can work in parallel, thus leading to either long training times or increased hardware requirements.

The third option, called minibatch processing, is a compromise between the two aforementioned approaches. Data is split into smaller parts that will comfortably fit into available memory - the minibatches - and the model is updated based on the gradient computed from these minibatches. While not offering the same favorable convergence performance as batch processing, the impact is kept much smaller compared to sample-by-sample processing and there is a considerable reduction of training time[10].

[10]The LSTM unit used in this work trained by more than factor four faster with minibatches of size 10 compared to batchsize of 1. 10 was the maximum number of minibatches that would reliably fit into the 11 Gb of vRAM of the nVidia GTX 1080Ti GPU used to accelerate training.

It is dictated by the nature of how RNNs consume data, that the data format is fundamentally different compared to FFNs. Where $\boldsymbol{x}$ for the MLP model in this work was a three-dimensional tensor of format with dimensions being time delay/feature and sample, in case of LSTMs, samples can not be concatenated along the time axis. The reason for this is that via means of the hidden state $\boldsymbol{h}$, the model forms implicit context to the data. Concatenating samples along the time axis would imply that data consists of a singular continuous time series. The discontinuities in the signal as well as the context belonging to the previous animal bleeding over into the next animal would lead to undesirable effects to the detriment of convergence towards a performant model and ultimately worse performance. As a consequence, data must be organized in a different way (section 4.1.2 and in particular, figure 4.5).

With this data format, concatenation must happen along a third axis, the batch axis, to comply with requirements dictated by the RNN nature of the model. This is however not possible out-of-the-box, since time axes of samples are of different lengths.[11] The solution for this problem is to pad all samples in a minibatch to the length of the longest sequence. *pytorch* offers a datatype called a *PackedSequence* that not only stores all the padded sequences but also the actual length of each sequence, masking the padded part at computation time so it is oblique to the model. After processing, the results have to be unpacked and flattened, afterwards a mask is applied to get rid of padding so that the loss function only considers valid data.

### 4.2.2 Multilayer Perceptron

#### Architecture

The MLP implementation for this work consists of 6 hidden layers á 1000 neurons, an input layer á 289 neurons[12] and a single output neuron. In between each layer and its activation function, batch normalization is used.[13]

[11] The reason for these dissimilar lengths lies in the nature of the data. Sensors are activated at different times for different animals and certain other data, particularly group activity data may not be available through the entire time, leading to time samples being dropped from the index.

[12] 288 time series data points and one for DIM information.

[13] The output layer is an exception.

### Objective Function

The most common choice of objective function for binary classification is the Binary Cross Entropy (BCE)/log loss function, which is defined as

$$l(x, y) = \frac{1}{N} \sum_{n=1}^{N} -w_n \left( y_n \cdot \log\sigma(x_n) + (1 - y_n) \cdot \log\Big(1 - \sigma(x_n)\Big) \right). \tag{4.1}$$

Training a model on data that contains a big class imbalance usually leads to the model classifying everything as the majority class, because this behavior helps it minimize the cost function sufficiently. Since this is not the intended behavior for a classifier, BCE can incorporate class reweighting, which makes the cost function consider one class much more heavily than the other. BCE is then defined according to

$$l(x, y) = \frac{1}{N} \sum_{n=1}^{N} -w_n \left( p y_n \cdot \log\sigma(x_n) + (1 - y_n) \cdot \log\Big(1 - \sigma(x_n)\Big) \right), \tag{4.2}$$

with $p$ being the weight of the positive class weight.

### Optimization

The optimizer used for training the MLP model is the *Adam* algorithm [KB15], which has become very popular in deep learning applications. *Adam* stands for *adaptive moment estimation* and is a stochastic gradient based optimization. The major novelty of *Adam* is that, as hinted at by the name, it adapts its learning rate based on estimates of the first and second moments of the gradient. This improves the convergence behavior, especially late in the learning phase and in cases where the gradient is very noisy.

#### Design considerations

The motivation for trying this type of model was not the expectation that it would perform exceptionally well. It started out as a baseline while the infrastructure around it was put in place. Eventually it achieved noticeable performance under controlled conditions.[14]

Compared to more specialized architectures, this type of neural network has an unfavorable way of scaling complexity wise regarding the number of input features. For example, if for each data point, the previous two days of time series data are considered as input features, this will result in 288 features, requiring 288 neurons in the input layer. If the classification task is multivariate, each additional time series that should be considered then adds an additional 288 input features for this configuration. Additionally due to the "general purpose" nature of a vanilla[15] MLP, learning a complex task such as multivariate time series classification requires individual hidden layers to be very wide as well, that is, to contain many neurons.

To reduce the complexity of the learning task and the required size of the input layer, this approach was used solely with the "heat feature" and the DIM as input (see section 4.1.3).

### 4.2.3 Long Short Term Memory

#### Architecture

For the LSTM architecture, a single LSTM cell with a cell width of 100 neurons was used. The hidden state is mapped onto the output via a fully connected layer with a sigmoid activation function.

[14] Particularly when the model did not have to deal with class imbalance. For more details see section 5.

[15] Vanilla in the sense of there being no specialization of layers towards a certain task.

### Input metrics

The input sequence is made up of four features, the activity, temperature with drink spikes removed, the DIM and the group activity. Both temperature and group activity are manually preprocessed metrics, therefore it is debatable whether this can still be considered a true deep learning task in accordance with the definition chosen for this work. However, in case of the group activity, using unprocessed signals would have been technically unfeasible. Groups consist of varying numbers of cows, which can not be accommodated, as the size of the feature space needs to be preset and can not be variable. Additionally feeding the group information to the model in this way would have inflated the feature space by one order of magnitude or even more and would most likely have required a larger model, which in term usually drives the requirement for data necessary to train. The resulting requirements with regard to difficulty of training as well as processing power could potentially have far exceeded the scope of this work.

### Objective Function

For the LSTM, same as for theMLP model, BCE was used (equations (4.1) and (4.2)) as loss function.

### Optimization

Again, same as the MLP model, the *Adam* optimizer was chosen.

### Gradient clipping

LSTM as implemented for this thesis displayed a tendency to suffer from a sporadically occurring exploding gradient. These rare events completely fouled the training process, driving model weights to regions from which recovery is not possible. To keep this from happening, gradient clipping was added to the model by clipping the gradient norm just before updating the model weights, as described in [GBC16].

# 5 Results

In this section metrics used to evaluate performance (section 5.1) as well as the experiments performed in the course of evaluation and their results are presented. Both the MLP and LSTM models were trained and tested via cross validation, the MLP on data with and without class imbalance (section 5.2), LSTM was only trained on unbalanced data[1] with two different class reweighting settings (section 5.3).

## 5.1 Evaluation metrics

Models are evaluated on the basis of four basic performance metrics:

- Precision
- Recall
- $F_1$-Score
- Accuracy

Each of these metrics except accuracy is calculated separately for both positive and negative class. Additionally the support of each class is calculated as well as the average and weighted average over classes.

The choice of using this ensemble of metrics for evaluation is well established for binary classification. Precision and recall together (when calculated for each class) give a much more complete view of the classification behavior of a model as opposed to accuracy alone, especially with regard to unbalanced data as is the case in this thesis. The reason for calculating both for each class separately

[1] Removing the class imbalance like for the MLP model would not have been possible in case of the LSTM, because it would have destroyed temporal context, which in case of the MLP was contained in a separate data dimension, see section 4.1.2.

is that neither considers true negatives. However, the true negatives of one class are the true positives of the other class, so this helps to cover a "blind spot" that could otherwise lead to an incomplete picture of performance.

The $F_1$-Score serves to combine both precision and recall into a single performance metric. Care must be taken in classification problems such as this not to rely solely on $F_1$, because it does not consider true negatives.[2] Also, in practice often either precision or recall is valued higher than the other, while the $F_1$-Score gives similar importance to both. Since the objective of this thesis is to evaluate a prototype and not a highly optimized production model, the $F_1$-Score is sufficient.

Additionally, a confusion matrix is presented for each experiment so the magnitude of true and false classifications relative to each other can be presented in an intuitive fashion.

### 5.1.1 Precision

The precision is the ratio of true positive cases over all positive classifications, defined as

$$\text{Precision} = \frac{tp}{tp + fp}. \tag{5.1}$$

$tp$ is the number of true positive and $fp$ is the number of false positive classifications. The significance of precision lies in stating how many of the positive predictions were actually correct. Precision ranges from 0 to 1, where 0 indicates that all positive predictions were wrong, while 1 indicates that all were correct.

### 5.1.2 Recall

Recall is the ratio of true positive classifications over the sum of true positives and false negatives, defined as

[2] This is somewhat remedied by calculating separate $F_1$-Scores for both classes.

$$\text{Recall} = \frac{tp}{tp + fn}, \tag{5.2}$$

with $fn$ being the number of false negative classifications. Intuitively, recall states how much of the positive labeled data was correctly classified by the model. It too ranges from 0 to 1, where 0 indicates that none of the positive cases were identified, while 1 indicates that all were classified correctly.

### 5.1.3 $F_1$-Score

The $F_1$-Score is defined as the harmonic mean of precision and recall:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5.3}$$

The $F_1$-Score is a positive number ranging from 0 to 1. Zero indicates that all classifications were wrong while 1 indicates that all classifications were correct.

### 5.1.4 Accuracy

The Accuracy reflects the overall agreement of model classification and label data and is defined as

$$\text{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn} \tag{5.4}$$

where $tn$ is the number of correct negative classifications. It is added for completeness, although not much emphasis should be put on it. Due to the class imbalance, even if none of the heats were correctly identified, their small number would cause accuracy to be quite high.

## 5.2 Multilayer Perceptron

Training for the MLP model was performed on a time slice of 2 days of the "heat feature" and the respective DIM value as input. Parameters for the *Adam* optimizer were as follows: learn rate = 0.01, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1 \cdot 10^{-8}$. These are *pytorch* defaults, except the learn rate was increased by one order of magnitude. Batch size was set to 50000. The validation split of data was performed with a testset ratio of 0.3 except when training on the dataset with included class imbalance. Training was performed over 50 epochs without shuffling of batches in between epochs.

### 5.2.1 Balanced Dataset

The dataset used in this work has a class imbalance factor of roughly 250, so for each sample that is labeled as heat, there are 250 samples labeled not as heat. This imbalance creates huge issues for a machine learning algorithm, or any classification algorithm in general, since few false negatives compared to the number of all samples will already result in a significant reduction of precision of the positive class. As a consequence, during early development of the MLP model, the dataset was artificially balanced by dropping a random selection of samples from the negative class until both classes were equal in numbers. The motivation for this experiment was to determine whether the model was rich enough for learning in an optimized setting, early on during development. Table 5.1 and figure 5.1 show the results.

| | **Precision** | **Recall** | $F_1$**-Score** | **Support** |
|---|---|---|---|---|
| no heat | 0.955931 | 0.919398 | 0.937309 | 2349665 |
| heat | 0.922466 | 0.957672 | 0.939739 | 2352823 |
| Accuracy | | | 0.938548 | 4702488 |
| Macro Avg | 0.939198 | 0.938535 | 0.938524 | 4702488 |
| Weighted Avg | 0.939187 | 0.938548 | 0.938535 | 4702488 |

Table 5.1: Test results with MLP model trained and evaluated on data with removed class imbalance.

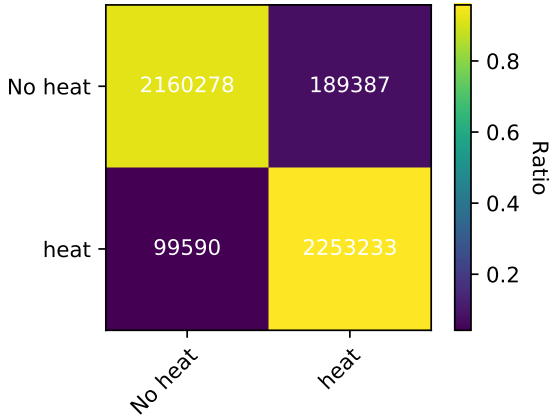


Figure 5.1: Confusion matrix of the MLP for the balanced dataset. A confusion matrix visualizes the accuracy of classification. Each row corresponds to a class label, while the columns correspond to the actual classification made by the model. As a consequence, the higher the values along the main diagonal and the lower the values along the secondary diagonal, the better the classification. The data for colorization is normalized per row.

Considering the superficial nature of the hyperparameter optimization that was done, the results are surprisingly good. For both classes, a $F_1$-score of about 0.94 is achieved. The confusion matrix (figure 5.1) underlines these results, the majority of values is situated in the main diagonal, which signals correct classification. The model was clearly able to capture the essence of what distinguishes both classes.

Still, it should be kept in mind that the balancing of classes removes a majority of the entire dataset, and as a consequence, a lot of opportunity for the model to make mistakes. The remaining experiments will unveil that the unbalanced dataset poses a much harder problem to solve.

### 5.2.2 Unbalanced Dataset

For the unbalanced dataset, two experiments were performed, one with the model trained on the balanced dataset and a second one with the model trained

on the imbalanced dataset, albeit with class reweighting in the objective function.

Results are presented in table 5.2 and figure 5.2.

| | **Precision** | **Recall** | $F_1$**-Score** | **Support** |
|---|---|---|---|---|
| no heat | 0.999838 | 0.937214 | 0.962211 | 474935652 |
| heat | 0.061904 | 0.969628 | 0.116379 | 2349381 |
| Accuracy | | | 0.927522 | 477285033 |
| Macro Avg | 0.530871 | 0.948471 | 0.539295 | 477285033 |
| Weighted Avg | 0.995221 | 0.927522 | 0.958048 | 477285033 |

Table 5.2: Test results with MLP model trained on data with removed class imbalance, evaluated on data with imbalance.

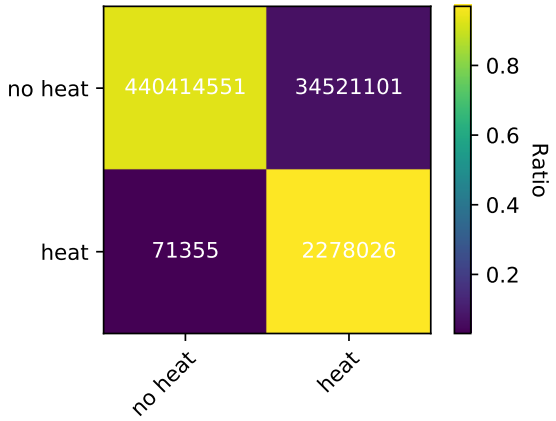


Figure 5.2: Confusion matrix for the MLP trained on balanced and evaluated on imbalanced data.

The results suddenly paint a very different picture compared to section 5.2.1. While the $F_1$-score for the negative class at 0.96 is rather good, for the positive class it sits at 0.11, owing to the very low precision of 0.06. The respective confusion matrix (figure 5.2) underlines the issue: The rows within themselves look good, but looking at the right column of the matrix, the number of false

positives (upper right) is an order of magnitude bigger than the number of true positives (lower right). This means that many samples were falsely classified as heats. This corresponds ti a large amount of false alarms. Owing to the significant class imbalance, the recall of just short of 0.94 for the negative class results in a number of false positives that dwarfs the total number of positive labels in the dataset.

Finally the model was trained on the dataset with imbalance present with class reweighting. The test-train split ratio had to be changed to 50%, otherwise the size of the training set would have exceeded available non volatile memory space. Positive class weight was set to 250.

The results are presented in table 5.3 and figure 5.3.

| | **Precision** | **Recall** | $F_1$**-Score** | **Support** |
|---|---|---|---|---|
| no heat | 0.999074 | 0.939812 | 0.968537 | 791556817 |
| heat | 0.063465 | 0.823969 | 0.117852 | 3918237 |
| Accuracy | | | 0.939241 | 795475054 |
| Macro Avg | 0.531269 | 0.881890 | 0.543194 | 795475054 |
| Weighted Avg | 0.994465 | 0.939241 | 0.964347 | 795475054 |

Table 5.3: Test results with MLP model trained and evaluated on data with imbalance.

The results echo what happened in the first experiment with unbalanced data. Due to the class imbalance, the number of false positives is much higher than the number of positive labels. There is a slight difference in that precision of the positive class is a bit higher while recall is 14 percentage points lower. The $F_1$-score for the positive class is nonetheless almost the same, so this experiment simply resulted in a different tradeoff between precision and recall, while the overall performance is not significantly different.

Despite working with class reweighting as well as training on the dataset with removed class imbalance, the MLP could not achieve the required performance. Specifically, the number of false positives remains an issue.

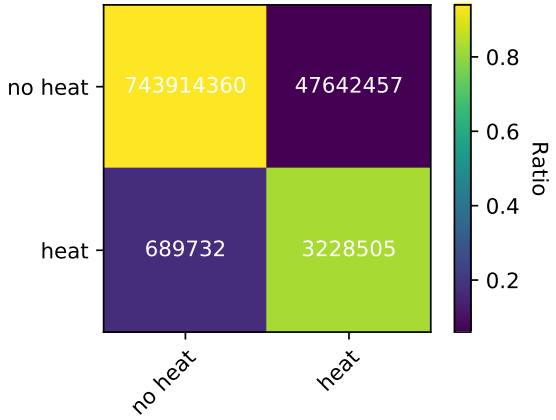


Figure 5.3: MLP confusion matrix for both training and evaluation on imbalanced data.

## 5.3 Long Short Term Memory

For the LSTM model, the parameters of the *Adam* optimizer were set to *pytorch* defaults (see section 5.2). Batch size was set to 10, testset size for cross validation was kept at a ratio of 0.3. Gradient clipping was set to 10. Input metrics used were DIM, activity, group activity and filtered temperature. Training was run for 50 epochs with random batch sampling.

The first of two experiments experiments was performed with a positive class weight of 100.

Results are listed in tables 5.4 and figure 5.4.

The theme from the MLP model continues with the LSTM model. A notable difference with a positive class weight of 100 is that the model even more heavily favors the negative class, resulting in even lower precision for the positive class, which should not come as a surprise, as it is a logical consequence of a lower class weight.

For the second LSTM experiment, parameters were kept similar except for positive class weight, which was set to 250.

Results are listed in tables 5.5 and 5.5.

| | Precision | Recall | $F_1$-Score | Support |
|---|---|---|---|---|
| no heat | 0.999421 | 0.840877 | 0.913319 | 528629444 |
| heat | 0.027132 | 0.901035 | 0.052678 | 2603609 |
| Accuracy | | | 0.841171 | 531233053 |
| Macro Avg | 0.513276 | 0.870956 | 0.482999 | 531233053 |
| Weighted Avg | 0.994655 | 0.841171 | 0.909101 | 531233053 |

Table 5.4: Test results for LSTM with class weight 100.

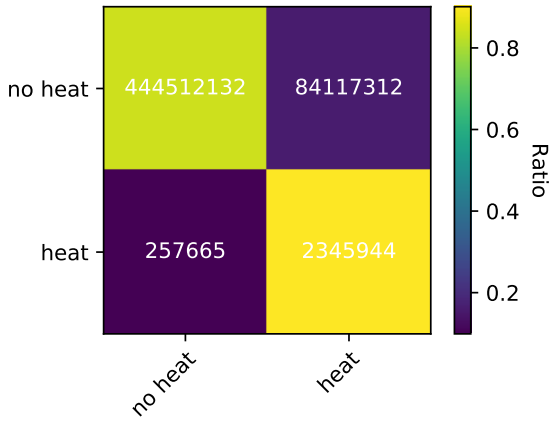


Figure 5.4: Confusion matrix for LSTM with class weight 100.

| | Precision | Recall | $F_1$-Score | Support |
|---|---|---|---|---|
| no heat | 0.999350 | 0.937664 | 0.967525 | 528629444 |
| heat | 0.064747 | 0.876201 | 0.120584 | 2603609 |
| Accuracy | | | 0.937363 | 531233053 |
| Macro Avg | 0.532049 | 0.906933 | 0.544054 | 531233053 |
| Weighted Avg | 0.994770 | 0.937363 | 0.963374 | 531233053 |

Table 5.5: Test results for LSTM with class weight 250.

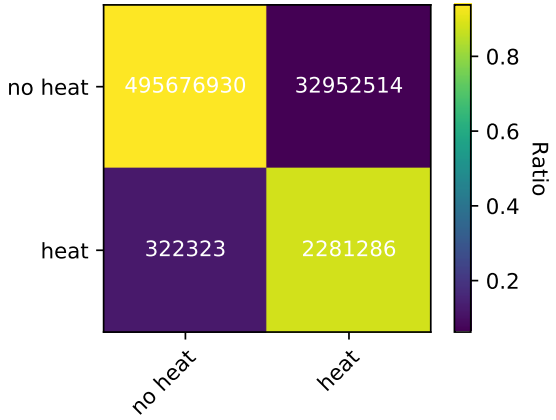


Figure 5.5: Confusion matrix for LSTM with class weight 250.

With this setting, the LSTM model is on par with the performance of the MLP model, even though the LSTM did not get the "heat feature", which contains both the information of activity and group activity as did the latter. The MLP model therefore did not have to learn the relationship between those raw inputs.

## 5.4 Summary

Looking at the results presented in sections 5.2 and 5.3, neither model was able to achieve the desired performance. Both do so in a remarkably similar way, by producing a high number of false positives, which in turn result in a low precision score of the positive class, since the support of said positive class is very low relative to the total number of samples. Neither training the MLP model with removed class imbalance nor using varying class reweighting in either model resulted in significant improvements.

To get to the bottom of why results are the way they are, chapter 6 looks at a couple of failure modes of the LSTM to offer explanations for why the models perform as they do and identify possible remedies.

# 6 Analysis of Classification Behavior

Looking at the results from chapter 5, apart from table 5.1 the results look rather underwhelming. There is a common theme of very low precision for the positive class, resulting in a mediocre $F_1$-Score. The generally acceptable scores for the negative class suggest that each model has simply learned to heavily favor the negative class. This is insofar surprising, as class reweighting was used to modify the cost function in a way that the positive class is represented according to the class imbalance factor.

Results for both MLP and LSTM models are suspiciously similar, suggesting that there may be some kind of underlying systematic issue. Given these circumstances, a closer survey of the classification behavior of the LSTM model trained with class weight of 250 was made to gain a better understanding. During the course of this survey, several points were identified that somewhat relativize the low positive class precision.

## 6.1 Classification Problems

### 6.1.1 Positive Class Label Temporal Precision

Section 4.1.3 discusses how labels are actually generated. The heat datetimes from meta data are used to seek local maxima of activity, around which, in a fixed timespan, positive labels are created. Figures 6.1 and 6.2 illustrate that this process does not always capture actual animal behavior. There are cases where a pre-heat may cause a premature onset of the activity increase. In other

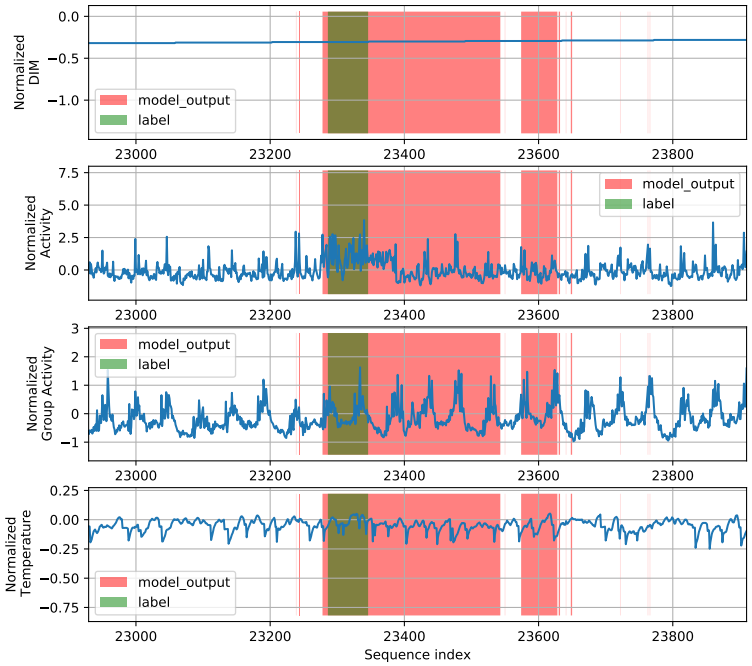


Figure 6.1: Example of label and LSTM model classification output overlayed on the input features. Red areas mark samples classified as heat by the model. Green areas mark samples labeled as heat.

cases, the local activity maximum does not always agree well with what a human expert would classify as the span of the heat.

These problems suggest that the approach used to generating heat labels is too crude. The lack of temporal precision in the labels causes the model to be unable to learn the underlying ground truth without making its heat classification timespan far too long. This factor alone causes an estimated number of false positives of two to three times the number of positively labeled data.

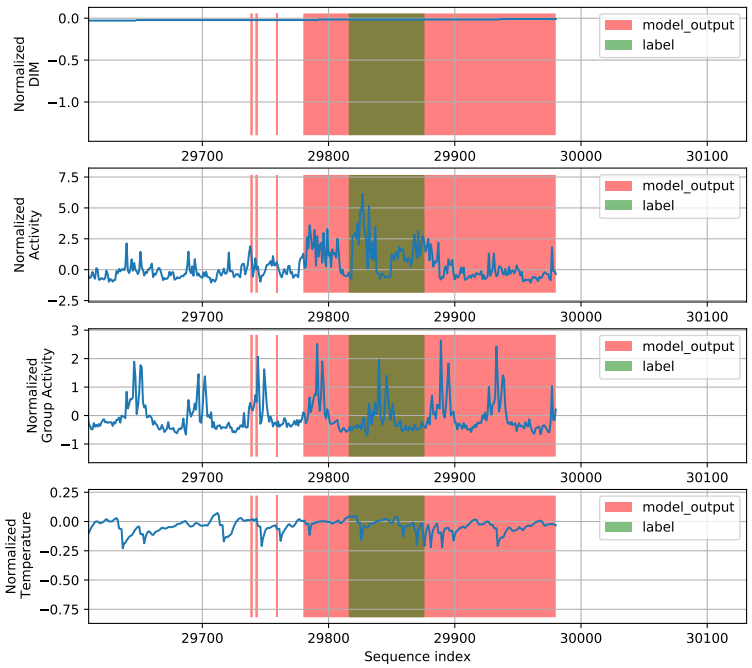


Figure 6.2: Similar plot to figure 6.1 from the same animal, at a different time. Note the distinct pre-heat not considered by the label.

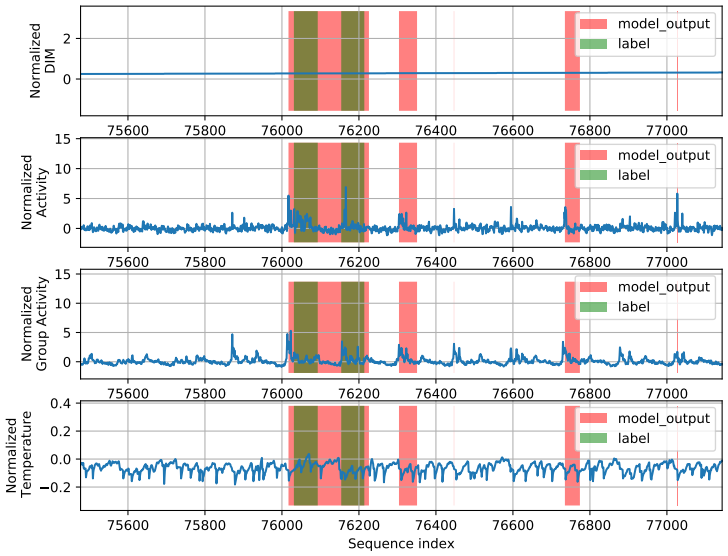


Figure 6.3: Numerous labels are in such close temporal proximity, that they must be considered duplicates. The above is a prime example of how these duplicates cause the model to widen the timespan of its heat classifications.

### 6.1.2 Duplicate Labels

While surveying the classification plots, it was discovered that a significant number of labels exist in close temporal proximity (figure 6.3). This serves as an aggravating factor to the problems caused by the lack of temporal precision of labels, causing a even stronger tendency of the model to make the timespans classified as heat overly wide. Overall, the ground truth used to train the models lacks temporal precision.

### 6.1.3 Classification Issues in Grazing Animals

There seems to be a strong influence from the way a farm operates onto how well the proposed methods are capable of detecting heats correctly. In particular it

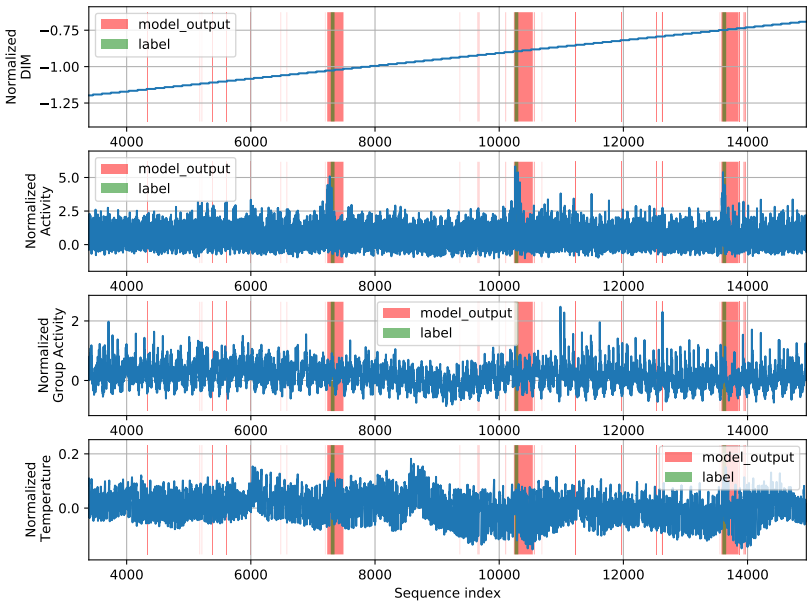


Figure 6.4: Example of a barn-held animal where classification worked considerably well.

is the non-stationary behavior of animals held outside that poses a fundamental problem.

Consider for example an animal held inside a barn (figure 6.4). Activity statistics are very stationary, resulting in excellent SNR. This in term makes it easy for the model to correctly recognize heats. On the other hand, the activity of the animal in figure 6.5 is highly non-stationary, suggesting that it spends varying amounts of time outside. The activity spikes displayed by the second animal result in numerous false positive classifications.

The overall behavior of the model therefore demonstrates that its classifications are heavily based on spikes in activity alone.

The extraordinary number of false positives in grazing cows indicates that it did not learn to consider the activity of the respective group, as information

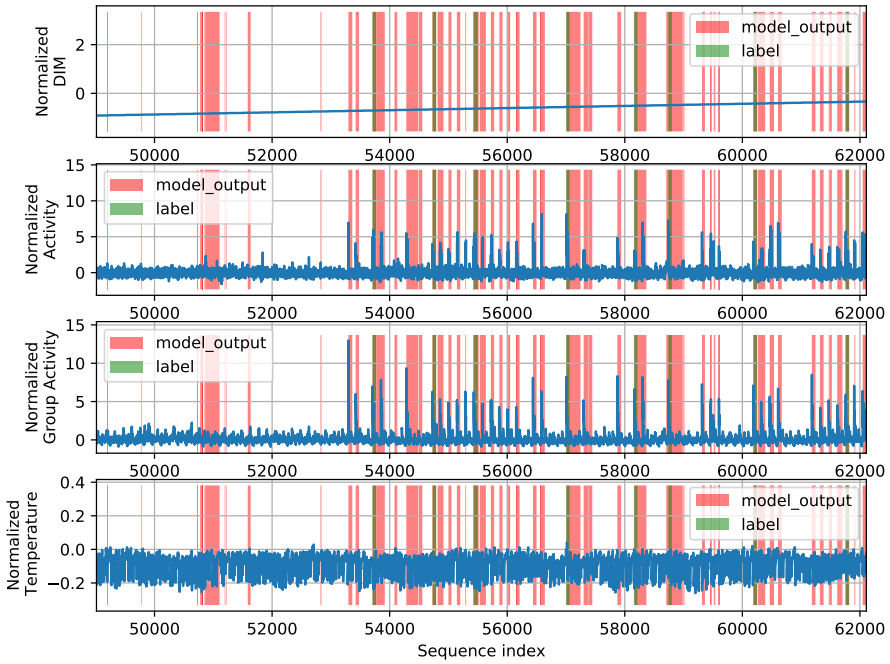


Figure 6.5: This animal is held on pasture sporadically. Note irregular activity spikes of varying magnitude that cause multiple false positives. These spikes are present in the group activity as well, which suggests that they were not caused by heats but instead by outside influence.

contained therein could aid the model in distinguishing in these cases. High activity being both present in the single animal and the group is a strong indication of the group being relocated. Further research should be put into determining whether the data might be heavily biased towards animals being held inside a barn as opposed to being held in pastures. If this is indeed the case, using techniques such as careful data augmentation to reduce the bias could help improve classification for pasture held animals.

## 6.2 Summary

Although there is a severe lack of precision for classifying heats, the approach prototyped for this work still shows promise. Several issues were identified that severely hinder the ability of the model to achieve proper classification. Despite these issues, the model has demonstrated the capacity to detect heats in barn-held cows reasonably well, even if lacking robustness. Chapter 7 will discuss some possibilities to improve on the existing system.

# 7 Outlook and future work

The goal of this thesis was to find a suitable architecture and implement a prototype machine learning model for dairy cow heat detection under a diverse set of livestock farming conditions. The milestone of reaching high overall model performance was not quite achieved. Designing a machine learning system is an iterative process however that does not conclude with implementing and testing the model. The experiments performed on the two models implemented in the course of this thesis revealed issues with the training data set. In particular, these are:

- Lacking temporal precision of labels
- Duplicate labels

Also an overly high false positive rate was discovered in animals with periodic spikes in activity, indicating that the models failed to learn to consider the group activity accordingly.

As a follow up to this thesis, future work will focus on improving the quality of the labels. Duplicate labels can be eliminated by using debouncing techniques, additional work will go into more careful placement of heat labels. These steps alone hold much potential for improvement.

Additionally, there is further potential in fine-tuning the models. Hyperparameter optimization was almost completely omitted in this work. Performance could benefit from tuning optimizer parameters as well as layer size and LSTM cell count. Analogous to batch normalization in FFNs, layer normalization techniques exist for RNNs [BKH16] as well that may improve convergence speed and quality.

Since the model was not able to learn the relationship between cow and group activity, either due to lack of training data size, model capability or both, a compromise could be to additionally feed the handmade “heat feature” or other

handmade features into the model that emphasize the single cow activity versus group activity correlation.

Due to the limited scope of this thesis, implementing the most promising architectures in the form of FCN and ResNet had to be omitted. Future work will include testing both architectures on the dataset, once the identified issues with labels have been solved.

Finally, recent research in the fields of language and sequence processing suggests that attention mechanisms such as used in the transformer architecture have the potential to be able to capture even longer distance sequence relationships than LSTMs. Future work should consequently consider implementing a transformer network for TSC in the hope of improving the overall performance.

# Appendix

# Appendix

# Acronyms

**ALSTM-FCN** Attention Long Short Term Memory Fully Convolutional Network
**API** Application Programming Interface
**BCE** Binary Cross Entropy
**BPTT** Back-Propagation Through Time
**CNN** Convolutional Neural Network
**COTE** Collective of Transformation-based Ensembles
**CPU** Central Processing Unit
**CSV** Comma Separated Values
**DFT** Discrete Fourier Transform
**DIM** Days in Milk
**DTW** Dynamic Time Warping
**FCN** Fully Convolutional Network
**FFN** Feed-Forward Network
**GPU** Graphic Processing Unit
**GRU** Gated Recurrent Unit
**HIVE-COTE** Hierarchical Vote Collective of Transformation-based Ensembles
**HDF5** Hierarchical Data Format version 5
**HMM** Hidden Markov Model
**ID** Identifier
**IOT** Internet of Things
**JSON** Java Script Object Notation
**k-NN** $k$-Nearest-Neighbors
**LSTM** Long Short Term Memory
**MLP** Multilayer Perceptron

**NaN** "Not a Number"
**RAM** Random Access Memory
**ReLU** Rectified Linear Unit
**ResNet** Residual Network
**REST** Representational State Transfer
**RNN** Recurrent Neural Network
**SGD** Stochastic Gradient Descent
**SNR** Signal-to-Noise Ratio
**SVM** Support Vector Machine
**TSC** Time Series Classification
**UTC** Universal Coordinated Time
**vRAM** Video Random Access Memory

# Bibliography


[Ani] smaXtec Animal Care GmbH Inside Monitoring. URL: https://smaxtec.com/ (cit. on p. 2).

[Bag+15] Anthony Bagnall, Jason Lines, Jon Hills, and Aaron Bostrom. "Time-Series Classification with COTE: The Collective of Transformation-Based Ensembles." In: *IEEE Transactions on Knowledge and Data Engineering* 27.9 (2015), pp. 2522–2535. DOI: 10.1109/TKDE.2015.2416723 (cit. on pp. 7, 8).

[Bag+17] Anthony Bagnall, Jason Lines, Aaron Bostrom, James Large, and Eamonn Keogh. "The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances." In: *Data Mining and Knowledge Discovery* 31.3 (2017), pp. 606–660. DOI: 10.1007/s10618-016-0483-9 (cit. on pp. 5, 7).

[Bah+16] Dzimitry Bahdanau, Jan Chorowski, Dimitriy Serdyuk, Philémon Brakel, and Yoshua Bengio. "End-to-end attention-based large vocabulary speech recognition." In: *Proceedings of 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2016, pp. 4945–4949. DOI: 10.1109/ICASSP.2016.7472618 (cit. on p. 19).

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML] (cit. on p. 68).

[CH67] Thomas Cover and Peter Hart. "Nearest neighbor pattern classification." In: *IEEE Transactions on Information Theory* 13.1 (1967), pp. 21–27. DOI: 10.1109/TIT.1967.1053964 (cit. on p. 7).

[Che+18] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. "Recurrent Neural Networks for Multivariate Time Series with Missing Values." In: *Scientific Reports* 8 (Apr. 2018). DOI: 10.1038/s41598-018-24271-9 (cit. on p. 20).

[Cho+14] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. "On the Properties of Neural Machine Translation: Encoder–Decoder Approaches." In: *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 103–111. DOI: 10.3115/v1/W14-4012 (cit. on p. 20).

[Chu+14] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. "Empirical evaluation of gated recurrent neural networks on sequence modeling." In: *Proceedings of NIPS 2014 Workshop on Deep Learning*. Dec. 2014 (cit. on pp. 20, 21).

[das] dask. URL: https://dask.org (cit. on p. 34).

[Dau+18] Hoang Anh Dau, Eamonn Keogh, Kaveh Kamgar, Chin-Chia Michael Yeh, Yan Zhu, Shaghayegh Gharghabi, Chotirat Ann Ratanamahatana, Yanping, Bing Hu, Nurjahan Begum, Anthony Bagnall, Abdullah Mueen, Gustavo Batista, and Hexagon-ML. *The UCR Time Series Classification Archive*. https://www.cs.ucr.edu/~eamonn/time_series_data_2018/. Oct. 2018 (cit. on pp. 7, 17).

[Fau+19] Kevin Fauvel, Véronique Masson, Élisa Fromont, Philippe Faverdin, and Alexandre Termier. "Towards Sustainable Dairy Management - A Machine Learning Enhanced Method for Estrus Detection." In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2019, pp. 3051–3059. DOI: 10.1145/3292500.3330712 (cit. on p. 23).

[Fir+02] Regina Firk, Eckhard Stamer, Wen Junge, and Joachim Krieter. "Automation of oestrus detection in dairy cows: A review." In: *Livestock Production Science* 75.3 (July 2002), pp. 219–232 (cit. on p. 29).

[Gas+15] Johann Gasteiner, Josef Wolfthaler, Wolfgang Zollitsch, Marco Horn, and Andreas Steinwidder. "Diagnostic validity of real time measurement of reticular temperature for the prediction of parturition and estrus in dairy cows." In: *Proceedings of 12th Conference on Construction, Engineering and Environment in Livestock*. Sept. 2015 (cit. on p. 1).

[GBB10] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks." In: *Journal of Machine Learning Research* 15 (Jan. 2010) (cit. on p. 10).

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016 (cit. on pp. 8, 12–14, 17, 20, 50).

[GS18] Omer Gold and Micha Sharir. "Dynamic Time Warping and Geometric Edit Distance: Breaking the Quadratic Barrier." In: *ACM Trans. Algorithms* 14.4 (Aug. 2018). DOI: 10.1145/3230734 (cit. on p. 6).

[h5p] h5py. URL: https://www.h5py.org (cit. on p. 32).

[He+16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Dec. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90 (cit. on p. 17).

[Hig+19] Shogo Higaki, Ryotaro Miura, Tomoko Suda, L. Mattias Andersson, Hironao Okada, Yi Zhang, Toshihiro Itoh, Fumikazu Miwakeichi, and Koji Yoshioka. "Estrous detection by continuous measurements of vaginal temperature and conductivity with supervised machine learning in cattle." In: *Theriogenology* 123 (Jan. 2019), pp. 90–99. DOI: 10.1016/j.theriogenology.2018.09.038 (cit. on p. 29).

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory." In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735 (cit. on p. 13).

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: *Proceedings of Machine Learning Research*. Vol. 37. July 2015, pp. 448–456 (cit. on p. 12).

[Ism+19] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre Alain Muller. "Deep learning for time series classification: a review." In: *Data Mining and Knowledge Discovery* 33.4 (2019), pp. 917–963. DOI: 10.1007/s10618-019-00619-1 (cit. on pp. 5, 7, 19).

[Jar+09] Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. "What is the best multi-stage architecture for object recognition?" In: *Proceedings of IEEE 12th International Conference on Computer Vision*. 2009, pp. 2146–2153. DOI: 10.1109/ICCV.2009.5459469 (cit. on p. 10).

[JJO11] Young-Seon Jeong, Myong K. Jeong, and Olufemi A. Omitaomu. "Weighted dynamic time warping for time series classification." In: *Pattern Recognition* 44.9 (2011), pp. 2231–2240. DOI: 10.1016/j.patcog.2010.09.022 (cit. on p. 6).

[Kar+17] Fazle Karim, Somshubra Majumdar, Houshang Darabi, and Shun Chen. "LSTM Fully Convolutional Networks for Time Series Classification." In: *IEEE Access* 6 (Dec. 2017), pp. 1662–1669. DOI: 10.1109/ACCESS.2017.2779939 (cit. on p. 19).

[KB15] Diederik P. Kingma and Jimmy Lei Ba. "Adam: A method for stochastic optimization." In: *Proceedings of 3rd International Conference on Learning Representations (ICLR)*. Dec. 2015 (cit. on p. 48).

[Kid77] Charles A. Kiddy. "Variation in Physical Activity as an Indication of Estrus in Dairy Cows." In: *Journal of Dairy Science* 60.2 (1977), pp. 235–243 (cit. on pp. 1, 29).

[LTB16] Jason Lines, Sarah Taylor, and Anthony Bagnall. "HIVE-COTE: The Hierarchical Vote Collective of Transformation-Based Ensembles for Time Series Classification." In: *Proceedings of 2016 IEEE 16th International Conference on Data Mining (ICDM)*. 2016, pp. 1041–1046. DOI: 10.1109/ICDM.2016.0133 (cit. on p. 7).

[Mar09] Pierre-François Marteau. "Time Warp Edit Distance with Stiffness Adjustment for Time Series Matching." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31.2 (2009), pp. 306–318. DOI: 10.1109/TPAMI.2008.76 (cit. on p. 6).

[NH10] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines." In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. 2010, pp. 807–814 (cit. on p. 10).

[OCo+11] Jared O'Connell, Frede Aakmann Tøgersen, Nicolas C Friggens, Peter Løvendahl, and Søren Højsgaard. "Combining cattle activity and progesterone measurements using hidden semi-Markov models." In: *Journal of agricultural, biological, and environmental statistics* 16.1 (2011), pp. 1–16 (cit. on p. 23).

[pan] pandas. URL: `https://pandas.pydata.org` (cit. on p. 33).

[pya] pyarrow. URL: `https://arrow.apache.org/docs/python` (cit. on p. 34).

[pyt] pytables. URL: `https://www.pytables.org` (cit. on p. 33).

[RK20] Marc Rußwurm and Marco Körner. "Self-attention for raw optical Satellite Time Series Classification." In: *ISPRS Journal of Photogrammetry and Remote Sensing* 169 (2020), pp. 421–435. DOI: `10.1016/j.isprsjprs.2020.06.006` (cit. on p. 21).

[Ros60] Frank Rosenblatt. "Perceptron Simulation Experiments." In: *Proceedings of the IRE* 48.3 (1960), pp. 301–309. DOI: `10.1109/JRPROC.1960.287598` (cit. on p. 8).

[SAD13] Alexandra Stefan, Vassilis Athitsos, and Gautam Das. "The Move-Split-Merge Metric for Time Series." In: *IEEE Transactions on Knowledge and Data Engineering* 25.6 (2013), pp. 1425–1438. DOI: `10.1109/TKDE.2012.88` (cit. on p. 6).

[skl] sklearn. URL: `https://scikit-learn.org` (cit. on p. 38).

[SLD17] Evan Shelhamer, Jonathan Long, and Trevor Darrell. "Fully Convolutional Networks for Semantic Segmentation." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.4 (2017), pp. 640–651. DOI: `10.1109/TPAMI.2016.2572683` (cit. on p. 17).

[sma] smaXtec. URL: `https://github.com/smaxtec/sxapi` (cit. on p. 32).

[Son+18] Huan Song, Deepta Rajan, Jayaraman J. Thiagarajan, and Andreas Spanias. "Attend and diagnose: Clinical time series analysis using attention models." In: *Proceedings of 32nd AAAI Conference on Artificial Intelligence, AAAI 2018*. Jan. 2018, pp. 4091–4098 (cit. on p. 22).

[Tan+16] Yujin Tang, Jianfeng Xu, Katsunori Matsumoto, and Chihiro Ono. "Sequence-to-Sequence Model with Attention for Time Series Classification." In: *Proceedings of 2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*. 2016, pp. 503–510. DOI: 10.1109/ICDMW.2016.0078 (cit. on p. 19).

[TD20] Christopher Tralie and Elizabeth Dempsey. *Exact, Parallelizable Dynamic Time Warping Alignment with Linear Memory*. 2020. arXiv: 2008.02734 [cs.SD] (cit. on p. 6).

[Tri+18] Trieu H. Trinh, Andrew M. Dai, Minh-Thang Luong, and Quoc V. Le. "Learning Longer-term Dependencies in RNNs with Auxiliary Losses." In: *Proceedings of 35th International Conference on Machine Learning, ICML 2018*. Vol. 11. Feb. 2018, pp. 7930–7939 (cit. on p. 20).

[Vas+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention is All You Need." In: *Proceedings of 31st Conference on Neural Information Processing Systems (NIPS)*. 2017 (cit. on p. 21).

[VV96] J.H. Van Vliet and Frank Van Eerdenburg. "Sexual activities and oestrus detection in lactating Holstein cows." In: *Applied Animal Behaviour Science* 50.1 (1996), pp. 57–69. DOI: https://doi.org/10.1016/0168-1591(96)01068-4 (cit. on p. 29).

[Wan+20] Shuilian Wang, Hongliang Zhang, Hongzhi Tian, Xiaoli Chen, Shujing Li, Yongqiang Lu, Lanqi Li, and Dong Wang. "Alterations in vaginal temperature during the estrous cycle in dairy cows detected by a new intravaginal device - a pilot study." In: *Tropical Animal Health and Production* 52.5 (Sept. 2020), pp. 2265–2271 (cit. on p. 29).

[WBS58] T. Randall Wrenn, Joel Bitman, and J. F. Sykes. "Body Temperature Variations in Dairy Cattle during the Estrous Cycle and Pregnancy." In: *Journal of Dairy Science* 41.8 (1958), pp. 1071–1076 (cit. on p. 29).

[WO15] Zhiguang Wang and Tim Oates. "Encoding Time Series as Images for Visual Inspection and Classification Using Tiled Convolutional Neural Networks." In: *Papers from the 2015 AAAI Workshop*. 2015 (cit. on p. 19).

[Wut] Matthias Wutte. *cattledb*. URL: https://github.com/wuttem/cattledb (cit. on p. 32).

[WYO] Zhiguang Wang, Weizhong Yan, and Tim Oates. "Time Series Classification from Scratch with Deep Neural Networks: A Strong Baseline." In: *Proceedings of 2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 1578–1585. DOI: 10.1109/IJCNN.2017.7966039 (cit. on pp. 17–19).

[YHL13] Ling Yin, Tiansheng Hong, and Caixing Liu. "Estrus detection in dairy cows from acceleration data using self-learning classification models." In: *Journal of Computers* 8.10 (2013), pp. 2590–2598 (cit. on p. 22).

[YK09] Lexiang Ye and Eamonn Keogh. "Time Series Shapelets: A New Primitive for Data Mining." In: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2009, pp. 947–956. DOI: 10.1145/1557019.1557122 (cit. on p. 7).

[zar] zarr. URL: https://zarr.readthedocs.io/en/stable (cit. on p. 34).